Clustered Multithreading for Speculative Execution

Rangsipan Marukatat

Doctor of Philosophy

Institute for Computing Systems Architecture

School of Informatics

University of Edinburgh

2003

Abstract

This thesis introduces the use of hierarchy and clusters in multithreaded execution, which allows several fragments of an application to be specifically optimised and executed by clusters of thread processing units (TPUs) as orchestrated by compile-time analysis. Our multithreaded architecture is a network of homogeneous thread processing units. Additional features were proposed, aiming at dynamic clustering of the TPUs throughout the entire program execution as well as minimum hardware support for speculative execution. The architecture executes a sub-set of the MIPS insruction set augmented with multithreaded instructions. A multithreaded compilation system was implemented, which focuses on high-level or front-end transformation from sequential C programs to multithreaded ones.

Empirical studies were conducted on benchmarks containing two types of program structures: loops and conditional branches. Coarse-grained control speculation enables simultaneous execution of several sub-problems such as loops, each of which could in turn be executed by multiple threads. Strategies were proposed for allocating TPU resources to these sub-problems and evaluated in simulations. Significant speedups were observed in the performance of multithreaded loop execution, and could be further improved by the application of control speculation.

Acknowledgements

I would like to thank my supervisor, D.K. Arvind, for his guidance and support throughout my study at the University of Edinburgh. The simulator and the compiler I used in my research were implemented, thanks to helps and suggestions from Alastair Patrick, Christoffer Arvidsson, and many others I talked to via the SUIF mailing lists.

I shared the office with Fang, Shun, and Grigori, whom I appriciate their company. My landlady, Deborah, and her Giant Schnauzers, Tavi and Talen, are always great friends and made me feel at home. I also thank for the friendship from many Thai friends, especially P'Noi, P'Fa, P'Ting, and Jay.

The biggest thanks go to my mom and dad for their love, support, and patience. Also, my little brother, Peune, who studied in France always sent jokes to me and our parents. I am very grateful to our relatives and friends in Thailand who looked after our parents while we were away.

My study was sponsored by a Thai Government Scholarship.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

There is a recent trend in multiprocessor architectures towards multithreading. Threads are streams of instructions with each one having its own program counter and register space. Whether the threads share memory space and other resources depends on the particular architecture and its implementation. A number of research groups have proposed architectural models which can be divided into two broad groups: Simultaneous Multithreading (SMT) and Chip Multiprocessing (CMP). The SMT-based model [3, 44, 45, 71] is built on a traditional wide-issue superscalar processor, which issues instructions from multiple threads to any available functional unit (FU) as the processor's resources are shared. The CMP-based model [26, 32, 58, 66, 70], which is analogous to a traditional tightly-coupled multiprocessor, fixedly partitions a single chip into multiple thread processing units (TPUs), each comprising of a number of functional units. The partitioning of computational resources (i.e. FUs) in SMT and CMP architectures are displayed in Figures 1.1(a) and (b), respectively.

Much effort has also been devoted to developing compilers for the multithreaded architectures, notably for CMPs [11, 32, 40, 53, 54, 66, 72, 79]. Unlike SMTs, which can exploit thread-level and instruction-level parallelism dynamically and interchange-

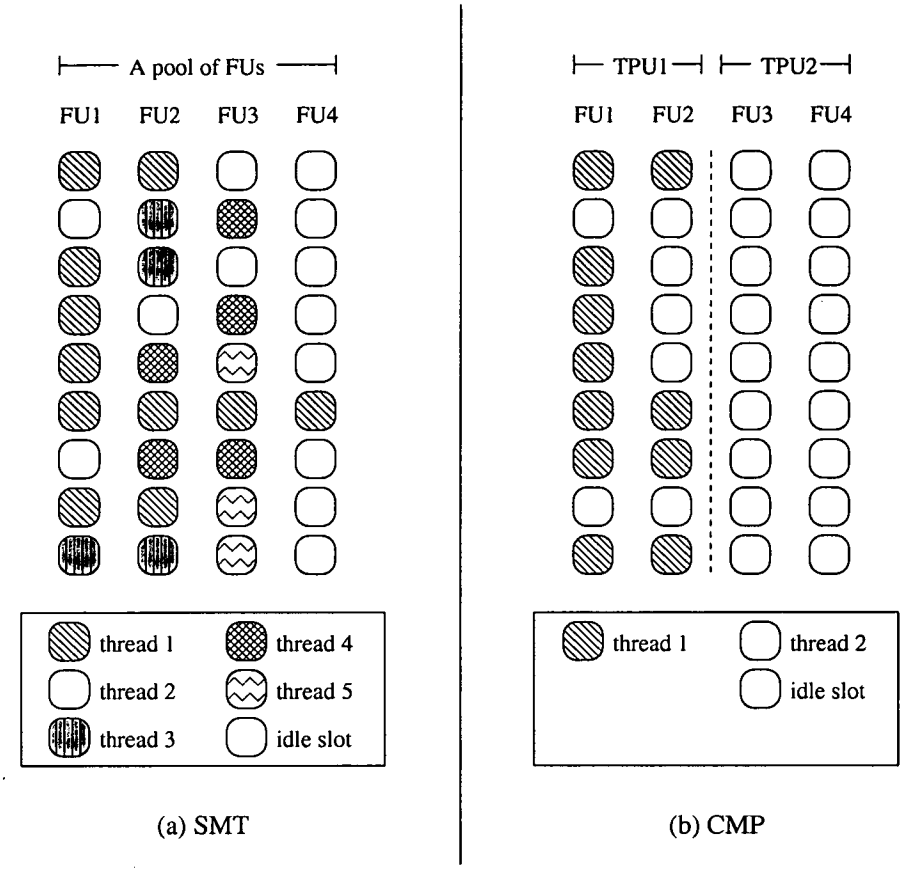


Figure 1.1: SMT and CMP architectures (reproduced from [44])

ably (i.e. in the absence of thread-level parallelism, an SMT would dedicate its resources for instruction-level parallelism), CMPs rely heavily on the compilers to extract thread-level parallelism and typically apply conventional optimisations to further exploit instruction-level parallelism. Because of this, however, CMP architectures are relatively simple to design and optimise compared to SMT ones.

1.1 Thesis Overview

This thesis proposes **a framework that organises the multithreaded execution on a CMP-based architecture into multiple layers or hierarchy**. The main ideas are:

- *Distributed program analysis* allow one to focus on classes of compilation techniques as well as resource requirement for each sub-problem, bearing in mind the overall constraints of the architecture.
- *Hierarchical thread management* alleviates the workload of overseeing and managing all threads in the global scope. Instead, groups of individual threads, corresponding to sub-problems, are mapped to clusters of TPUs and managed locally.
- *Dynamic clustering of the TPUs* enables resource allocation to be adjusted to specific requirements of the sub-problems during the program execution.

Hierarchical program partitioning is employed. Firstly, a program is divided into a (small) number of subsystems which are, for example, paths of conditional branches or outermost loops. They can be repeatedly decomposed into finer subsystems. Eventually, the innermost or the deepest ones are individual threads. Clusters of TPUs are allocated to the program partitions and their sizes depend on the inherent parallelism in those partitions. To enable this, a CMP-based architecture is provided with the ability to construct and manage clusters at run-time, as dictated by the compile-time analysis. An interface that conveys commands and inquiries between the compiler and the architecture is a set of special instructions added to the standard MIPS instruction set [22, 37]. An overview of the framework is shown in Figure 1.2.

Figure 1.3 depicts an example of program partitioning. The hierarchy is managed through *master/slave* relationships between threads, i.e. a cluster which is manipulated

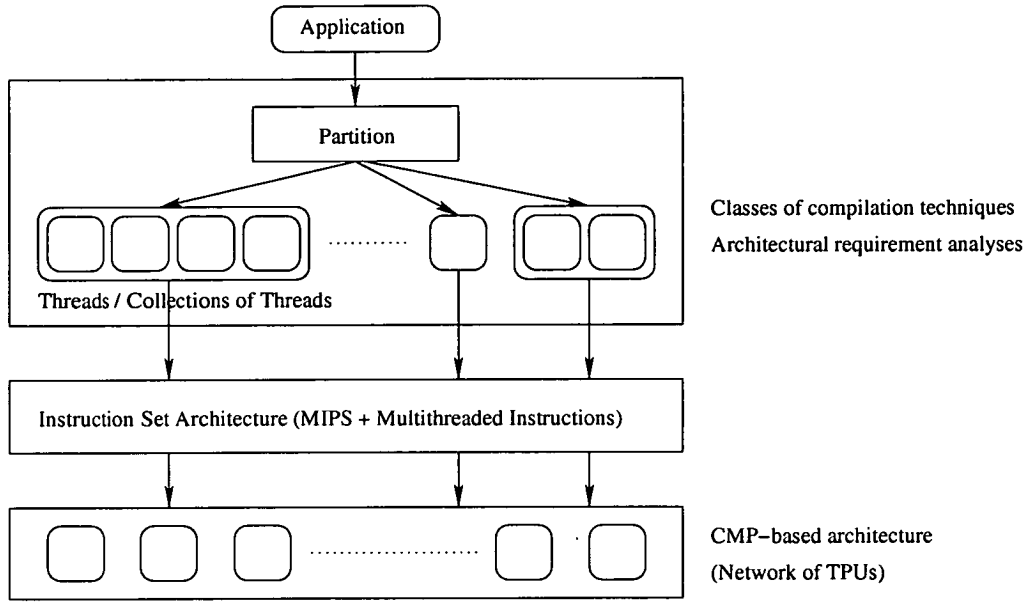


Figure 1.2: The system overview

by the master thread is allocated to a collection of slave threads, while each thread in the collection could, in turn, be the master thread of another cluster, and so on. The number of TPUs required in each cluster is determined at compile-time. At run-time, there could be both independent threads and collections of threads running on TPUs or clusters of TPUs. There are two levels of resource competition: (1) the master threads compete for the available TPUs in order to form clusters; and (2) the threads within the group allocated to a cluster compete for the available TPUs within the cluster.

The assignment of clusters of TPUs to collections of threads is illustrated by analogy with the assignment of clusters of FUs to threads of code in the SMT model. Both share an underlying idea that the resource partitioning and assignment are dynamically performed throughout the program execution rather than fixedly done in the hardware. In the SMT model (Figure 1.1(a)), the FUs are virtually clustered and de-clustered by

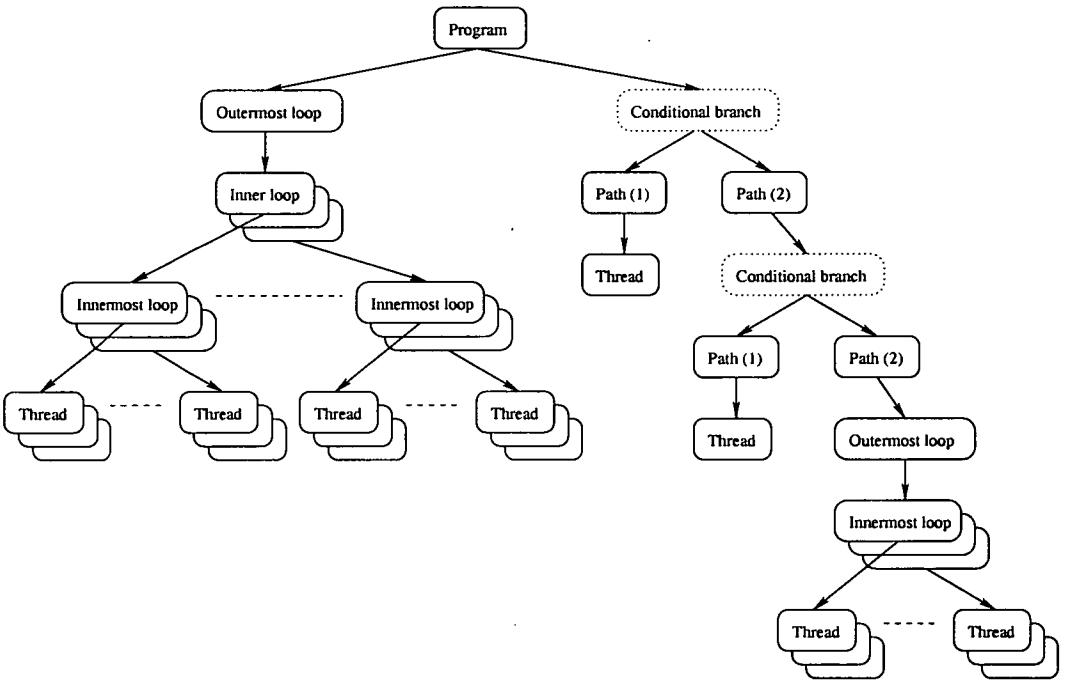


Figure 1.3: An example of program partitioning

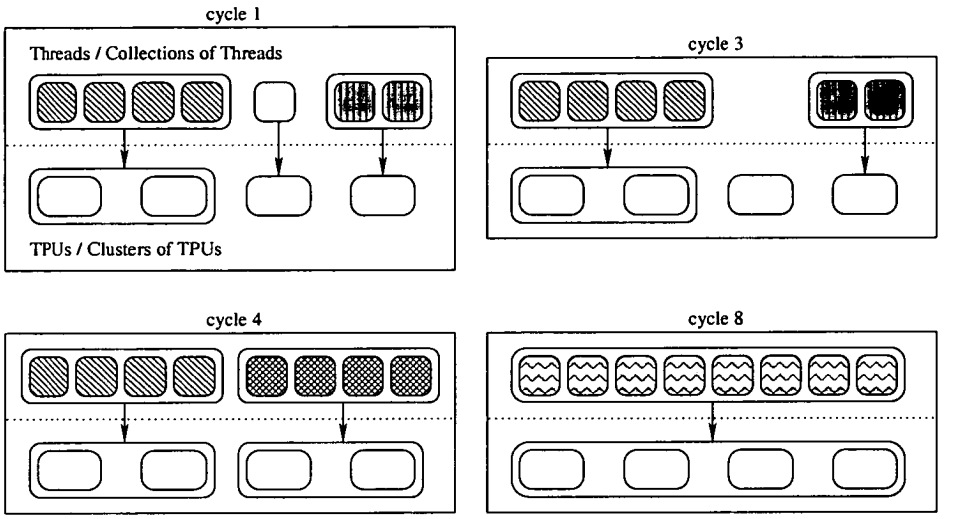
threads on a cycle-by-cycle basis. In other words, multiple threads compete for the FUs in each cycle. The number of FUs used by each thread depends on the instruction-level parallelism and the availability of resources, both of which are exposed at run-time.

In our model (Figures 1.4(a) and (b)), multiple collections of threads are generated to execute program partitions and compete for the TPUs. The number of TPUs executing each program partition depends on the thread-level parallelism predicted at compile-time and the availability of resources known at run-time. Figure 1.4(a) displays snapshots of the program execution shown in Figure 1.4(b). *Clusters* {*1, 2, 3, 4, 5*} are allocated to *collections of threads* {*1, 2, 3, 4, 5*}, respectively. At *cycle 1*, there are 3 program partitions being executed simultaneously, one by a cluster of 2 TPUs and the rest by a single TPU each. During the execution of each program partition,

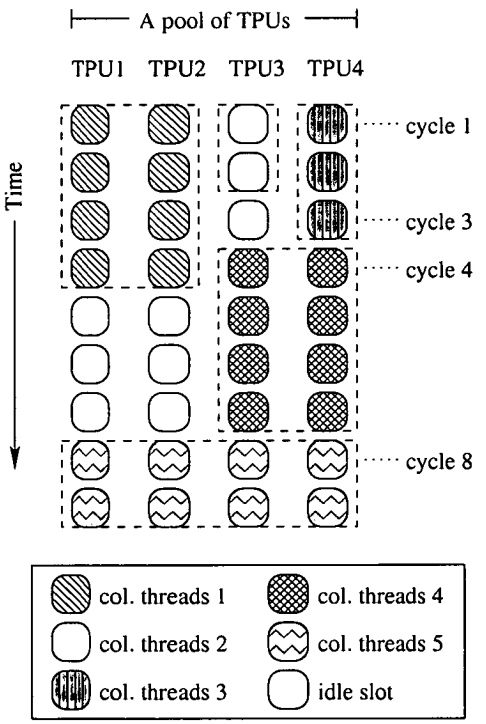
multiple threads may reuse a TPU since they may be spawned and retire at different cycles. The threads spawned concurrently can only compete for the available TPUs in the cluster allocated. At *cycle 4*, *cluster 1* is still active while *cluster 2* and *cluster 3* have freed their TPUs which are grabbed by *cluster 4*. At *cycle 8*, only *cluster 5* is active which uses all the available TPUs.

An advantage of dynamic cluster allocation is in the utilisation of TPU resources by various sub-problems in the program. For instance, if a non-speculative and a speculative loops are to be executed in parallel, a small number of TPUs should be dedicated to the speculative loop while the rest are reserved for the other computation. This approach differs from other clustered multithreaded architectures (e.g. [21, 38, 47, 78]) in that the others statically allocate clusters, as shown in Figure 1.4(c). Within the clusters, their resource partitioning could be in either SMT [38, 47] or CMP [78] style.

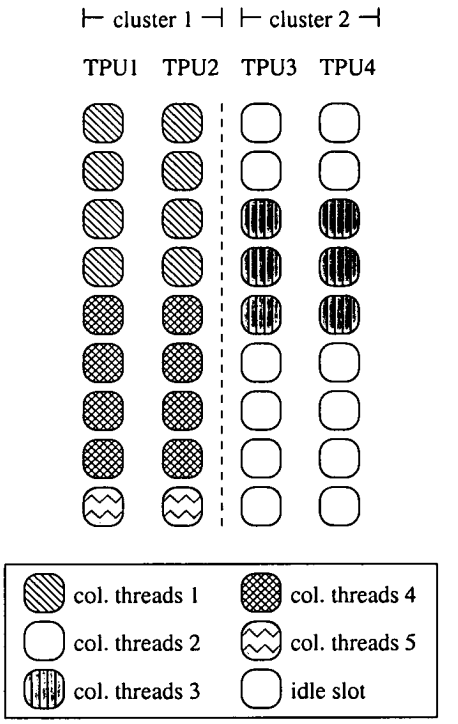
The main contribution of this thesis is the experimental evaluation of hierarchical multithreading in a framework consisting of a simulated multithreaded architecture and a compiler. The focus is on two types of program structures: loops and conditional branches. Loops are potential sources of parallelism and their nesting structures fit well with the hierarchy. Control speculation is a well-known method for exposing parallelism in programs although the speculative execution is not guaranteed to be useful. Based on the experimental results, significant program speedups were achieved by loop parallelisation, and could be further improved by control speculation.



(a) Snapshots of (b)



(b) Our approach



(c) Conventional approach

Figure 1.4: The clustering of TPUs during program execution

1.2 Thesis Organisation

The remaining chapters are summarised as follows:

Chapter 2 reviews issues concerning multithreaded execution such as (1) creation, initialisation, and retirement of individual threads; (2) interaction between threads such as communication, synchronisation, and thread-level speculation; and (3) their collective relationship in clusters and hierarchy.

Chapter 3 describes the multithreaded architecture, which is based on a CMP processor similar to the Superthreaded architecture [68, 69, 70]. It was enhanced to support hierarchical execution, control speculation, register synchronisation and forwarding, and novel multithreaded instructions. The multithreaded compiler implemented using the SUIF package [84] is also described. It takes advantage of well-defined intermediate representation to recognise and transform loops and conditional branches in sequential programs for multithreaded execution.

Chapter 4 presents examples of multithreaded loop execution. Transformation routines implemented in the compiler are described, followed by experimental results and discussion.

Chapter 5 presents examples of multithreaded control-speculative execution. It describes how programs are transformed and executed. Strategies used to partition programs for control speculation and to allocate resources are explained. Experimental results are presented and discussed.

Chapter 6 summarises and discusses the main findings of this research with suggestions for future work.

Chapter 2

Literature Survey

The key ideas in multithreaded execution are as follows:

1. The creation, initialisation, and retirement of individual threads.

2. The interaction between threads, essentially the inter-thread communication, synchronisation, and thread-level speculation.

3. The collective relationship, such as hierarchical organisation and clustering.

We examine these ideas in some well-known multithreaded architectures, such as *Single-Program Speculative Multithreading (SPSM)* [18], *Superthreaded* [68, 69, 70], *Stanford Hydra* [31, 32, 53, 54], *CMU STAMPede* [65, 66, 67], *Multiscalar* [12, 26, 35, 72], *Trace processors* [58, 59, 60], *UPC Speculative Multithreaded* [45, 46, 47], and *Dynamic Multithreading (DMT)* [3]. SPSM, Superthreaded, Hydra, and STAM-Pede combine various software and hardware techniques. Multiscalar relies heavily on the hardware although compiler assistance is still needed. On the other hand, Trace, Speculative Multithreaded, and DMT are solely hardware-based.

2.1 Thread Creation

2.1.1 Dynamic Approach

UPC Speculative Multithreaded, DMT, and Trace Processors use different criteria to extract multiple threads from sequential programs.

The UPC Speculative Multithreaded detects loops at run-time and generates threads to execute the loop iterations concurrently. In [45, 46], a single fetch stream mechanism was implemented. Instructions are fetched from the same program counter and broadcast to all the threads. In their follow-up work [47], a *loop trace* was introduced to support multiple control-flows. Each entry in the loop trace is a sequence of the predicted branch directions that defines a particular control-flow.

DMT creates threads at procedural and loop boundaries. An *after-call* thread executes the instruction at the static address after the call, while the parent thread enters the procedure body. Likewise, an *after-loop* thread starts its execution at the static address after the loop. Although this lookahead technique exploits coarse-grained parallelism, it suffers from poor resource utilisation. Because threads are spawned in the reverse program order, the most recently-created threads are the earliest ones to retire. The oldest threads which are typically further away from the main execution point hold resources for a longer period before retiring. To solve this, an *adaptive thread predictor* assigns priority to threads using the lookahead distance and history patterns. The threads with higher priority will pre-empt the ones with the lower priority.

Trace processors construct traces from the dynamic instruction stream. The trace size is restricted to 16 instructions, or even shorter if any call indirect, jump indirect, or return instruction is encountered. Traces are stored in the *trace cache*. The *next-trace predictor* [36] predicts the next instruction sequence and looks in the trace cache. If

the trace is found, it is fetched and sent to the processing unit. Otherwise, the trace is constructed by fetching from the instruction cache.

2.1.2 Static Approach

SPSM supports the *master/slave* model. The program execution starts with the main thread. It forks new threads which are ahead of itself in the program order. The threads are merged when the main thread reaches the starting address of the future thread and the future thread encounters a *suspend* instruction. After merging, the main thread resumes the execution after the *suspend.* SPSM is unaware of the actual resources at run-time. Depending on the hardware implementation, a thread may or may not be successfully forked. Hence the correct program execution must be preserved whether each code region is executed by the main thread (fork fails) or a future thread (fork succeeds).

The Superthreaded compiler partitions a program into threads and each thread into four pipeline stages. Continuation variables such as loop index variables are computed in the first stage as they are needed for sparking a new thread. The next stage computes target store addresses and forward them to the successors for run-time checking of data dependencies. The main computation and data communication is performed in the following stage. Finally, the thread is synchronised and commits data to the data cache before retiring. Their thread allocation policy is to delay forking until the next thread processing unit is available, while the current thread continues after the fork instruction without stalling. Because of this, the Superthreaded's performance is likely to be sensitive to the workload distribution among threads.

Hydra supports two types of parallel threads: subroutine (after-call) threads, and loop iteration threads. The subroutine threads are created automatically at run-time

when procedure calls are encountered. However compiler support is needed to identify potential loops and perform source-to-source transformation for speculative parallel execution. Threads are manipulated at run-time by software exception handlers which are implemented in the speculative coprocessor. STAMPede's approach is very similar to Hydra's. A program is partitioned into units of execution, *epochs*, at compile-time and the software handling routines manage threads at run-time.

Unlike SPSM, Superthreaded, Hydra, or STAMPede, Multiscalar is biased toward extensive hardware support for inter-task register communication, and control and data speculation. However, it still relies on the compiler to analyse the control-flow graph of a program and use heuristics to group basic blocks into tasks. A task descriptor is generated for each task to indicate its boundary, a list of possible successor tasks for the run-time control-flow speculation, and the inter-task data dependence information.

In Hydra and STAMPede, the partial ordering between threads or epochs can be controlled by the compiler, by passing the thread/epoch number as an argument to the fork routine. In Multiscalar, the task identification number is read from the task descriptor. In Superthreaded, since a new thread only starts on the next thread processing unit in the uni-directional ring, the thread ordering is implicitly known by the order of the thread processing units and the *head thread* pointer.

2.2 Thread Initialisation

When a new thread is sparked on a processing unit, its program counter is set to the address it will start the execution. Local components in the processing unit, such as register file and branch predictor, are initialised as described next.

2.2.1 Register Context

The most common approach is to copy the current register values from the parent's register file to its child's [3, 18, 45, 46, 58, 60]. Based on dataflow definitions given in [2, 4], a register carries a *live-in* value at the beginning of the child thread if the child thread reads from this register before any writes to it. Also, since this register carries a *live-in* value to the child, it is considered to carry a *live-out* value from the parent. At the time of forking, some registers may not yet be available. There are two ways to handle this:

1. Enforce synchronisation in the child thread until the values are produced and forwarded from the parent.

2. Use value prediction techniques to speculate the live-in values.

To enforce synchronisation in the child thread, the compiler may explicitly insert synchronisation primitives such as *barrier* or *blocking receive* before the instructions that consume the live-in values. In Multiscalar [12], a *create mask* is read from the task descriptor, which identifies all registers that may be written during the task execution. The task also receives an *accum mask* from its parent, which is the accumulation of the *create masks* of all the active predecessors. It will block if it tries to use the registers indicated in the *accum mask* whose values have not yet been received.

Architectures that opt for the live-in value speculation include DMT, UPC Speculative Multithreaded, and Trace processors. DMT allows the child thread to speculatively copy all the current values from its parent at the spawning point. Because the lookahead policy spawns threads which are further away from the current execution point, it might incur a high misprediction rate, particularly for after-loop threads. On the other hand, there is often false data dependence due to register saves and restores in the pro-

cedure call sequence. Value prediction for after-call threads is likely to be beneficial. Their experiments on the Spec95 benchmarks show a significant prediction accuracy; however, most benchmarks perform better when only after-call threads are allowed.

The UPC group [45, 46] uses execution history from an *iteration table* to determine register predictability. The hardware initialises predictable live-in registers for a new thread by inserting *add $R, $R, stride* instructions at the beginning of the dynamic instruction stream. Unpredictable registers are mapped to the live-in register file. The child thread will stall if it tries to read those registers, until they are forwarded from the parent.

In the Trace processor, before a trace is stored in the trace cache, it is preprocessed in the hardware by identifying local, live-in, and live-out values. When the trace is fetched and started on a processing unit, it receives predictable live-in values from the value predictor, whereas unpredictable values are obtained from the global register file during the trace execution.

Finally, in Krishnan and Torrellas [39], when a new thread is initialised on a processing unit, some registers in the local register file are invalidated while the rest (with existing values) are reused by the new thread.

2.2.2 Branch Predictor

There are at least three options for initialising the local branch predictor:

1. Copy the branch history table from the parent thread. This approach incurs a higher initialisation overhead than the other two. A study by Marcuello and Gonzalez [49] showed that it gave a very close performance to the gshare predictor in the single-threaded execution, which predicts a branch by using the combined history of all the recent ones. The branch address and the combined history are

exclusively-ORed (XORed) to form an index for accessing the prediction table in the gshare.

2. Use the current state of the branch history table as it was left by the previous thread executing on this processing unit. This option could incur less prediction accuracy due to more arbitrary branch correlation between the previous and the current threads. Marcuello and Gonzales [49] also showed that this option suffered at least a 10% performance degradation.

3. Initialise the branch history table to some fixed values, such as 0. With this approach, early branches in the thread have no memory from the previous execution. As the thread proceeds, the branch history is built up for later branches. An experiment by Akkary [3] showed that this scheme performed as well as the gshare predictor in the single-threaded execution.

2.3 Thread Retirement

Multithreaded execution can be broadly categorised into master/slave and predecessor/successor models. Conditions as to when and how threads in these models retire, update program's state, or handle exceptions are different, as described next.

2.3.1 Master/Slave Model

In this model, the master thread maintains the state of the program. It forks slave threads to execute instructions which are ahead of itself in the program order. At some point, e.g. when a slave completes its execution, it will be merged into the master thread. The *merge* action typically induces an effect as if the slave's execution has been

performed by the master itself. For instance, in the SPSM architecture, the register values updated by the slave are copied back to the master's register file. The master also receives the updated program counter and consequently resumes the execution after the last instruction executed by the slave. An exception raised by the slave will be delayed and handled after it has been merged into the master.

2.3.2 Predecessor/Successor Model

In this model, a sequential order of active threads is maintained. The *head thread* which is the first thread in the list represents the current state of the program. It is usually the only *non-speculative* thread while the others could be speculative. When the head thread finishes its execution and retires, the next thread in the order list becomes the new head thread and its state becomes the current program state. Generally, if a thread causes an exception, it will stall until it becomes the head thread. Then the instructions before the one that raised the exception are retired and the exception handling is processed. As mentioned in [31], if the stalled thread is mispredicted and aborted, the exception should be discarded because it would not have occured in the sequential execution.

Steffan et al. [65] use software interface to emulate the predecessor/successor model, which is called *one-shot threading*. Instead of relying on a centralised hardware structure, the identification of the oldest and the least speculative epoch is controlled by the software, by passing a *homefree token*. An epoch can be forced to block until it receives the homefree token. It can then commit the speculated results, pass the token to the next epoch, and retire.

2.4 Inter-thread Data Communication

Threads communicate data in the initialisation phase and during their execution. The communication between threads can be categorised as follows:

1. Producer-driven. Producers initialise the communication, such as register forwarding in Multiscalar.

2. Consumer-driven. Consumers initialise the communication.

3. Producers and consumers communicate via shared medium such as global register files or shared memory.

The register communication in Multiscalar is local reads/distributed writes, i.e. an instruction reads a register value from the local register file and, if tagged with a *forward bit*, propagates the value it produces to successor tasks. Vijaykumar [72] proposed *register communication scheduling* techniques targeted at the Multiscalar architecture. He studied four strategies for register communication: *End_send* forwards all registers at the end of the task execution; *Eager_send* forwards a register every time it is modified; *Last_send* forwards a register after its last modification; and *Spec_send* forwards a register when there is a high probability that it will not be modified again. The first two strategies do not require any compiler support, whereas the others require dataflow analysis to determine the last modification of each register. Eager_send and Spec_send also involve squashing threads and re-forwarding the values.

Traces in the Trace processor communicate via a global register file. During the execution, the producer trace sends live-out values to global result buses, whereas the consumer reads from the global register file or monitors the buses.

Superthreaded forwards memory data instead of registers. A thread computes target store addresses and passes them to its successor. The successor will stall if it tries

to load from these addresses before the data is made available. As soon as the predecessor stores data in its own memory buffer, the data and the store address will be forwarded to the next thread.

2.5 Synchronisation

There are computations such as reduction operations in which the ordering of threads is irrelevant; however, only one thread should be allowed to update shared data at any time. This section focuses on two types of synchronisation to handle this situation: code locking and data locking.

Code locking permits one thread at a time to execute the code inside the critical section. Common synchronisation techniques include *mutex locks*, *conditional variables*, and *semaphores* [14, 41]. The synchronisation variables used in all these techniques are stored in global registers or shared-memory areas. Architectures that support speculation may allow only non-speculative threads to execute the critical section, as suggested in [68]. The restriction prevents speculative threads from impeding the non-speculative ones.

At a fine-grained level, data locking enforces synchronisation on data items. A widely-used technique is *multiple readers/single writer* locks [14, 28, 50]. There are three variations to this scheme: reader preference, writer preference, and fair lock. All of them require readers to block until the current writer finishes. With a reader preference lock, once there are readers currently active, new readers that arrive can proceed even though there is a writer waiting. Conversely, with a writer preference lock, the current readers are suspended if a new writer arrives. In the case of a fair lock, new readers wait until earlier writers finish, while a new writer waits until both

Table 2.1 Categories of thread-level data speculation

Type	Register Related	Memory Related
value	register values	memory load values
dependence	register communication	memory references

readers and writers before it finish. Furthermore, many multiprocessors support atomic read-modify-write operations such as test-and-set and fetch-and-op.

An alternative lock-free technique uses a pair of *load-linked* and *store-conditional* instructions [7, 57]. A thread executes a load-linked instruction to load an original value from a memory location, performs further computation, and tries to store a new value back using a store-conditional instruction. The load-linked approach does not prevent the other threads from loading the data or executing the critical code following it. However, only one thread will successfully store the new data back to the memory. The others whose store-conditionals failed may retry the computation.

2.6 Thread-Level Speculation

Thread-level control speculation enables threads to *start* execution before the conditions on which they are dependent are resolved. On the other hand, thread-level data speculation enables threads to *continue* the execution in spite of data dependence between concurrent threads. It is further categorised as illustrated in Table 2.1. Value speculation speculates on register or memory load values. Memory dependence speculation conventionally speculates in the midst of ambiguous memory references. Finally, register dependence speculation assists inter-thread register communication.

2.6.1 Control Speculation

In SPSM, Superthreaded, Hydra, and STAMPede, thread-level control speculation is performed by the compilers. In Multiscalar, tasks and their associated task descriptors are generated at compile-time. At run-time, the global sequencer predicts the next task which is one of the possible successors indicated in the current task descriptor.

Both the task predictor in Multiscalar and the trace predictor in Trace processors are based on *path-based trace predictors* proposed by Jacobson et al. [35, 36]. Clustered Speculative Multithreaded [47] uses a *loop trace* which is also adapted from Jacobson's to predict control flows of the loops containing multiple conditional branches. An *adaptive thread predictor* in DMT assigns priority to threads using criteria such as lookahead distances and global history.

Misspeculation penalty at the thread level can be higher than in case of the individual branch prediction. Because the predicted branch is usually the last instruction in the thread, it takes many cycles before the branch is finally resolved and the wrong thread is squashed. To keep the misspeculation penalty as low as possible, many architectures and compiler techniques include low-confident branches within the threads and expose high-confident branches to the thread-level speculation. In practice, the embedded branches may have even lower predictability than when they are predicted in the sequential execution. This is because the local branch predictors do not have a complete view of the continuous (global) dynamic instruction stream.

The point where both paths of a conditional branch rejoin indicates the start of the *control-independent path* of that branch. Since the control-independent path will be executed regardless of the outcome of the branch, another thread can be launched to execute this path in parallel with the main and the control-speculated threads. The *control-independent thread* must also be treated as a speculative thread because it may

still be data dependent on either path of the branch. This aspect of control independence has been studied in detail by Rotenberg [59, 60].

On the other hand, there are works such as *Thread Multiple Path Execution (TME)* [73] and *Selective Dual Path Execution (SDPE)* [33] that allow the execution of both paths of the hard-to-predict branches. TME spawns threads to execute the less likely paths when there are fewer threads running than the available hardware contexts. SDPE investigates dual-path forking policies in detail.

2.6.2 Register Speculation

Well-known value speculation techniques in superscalars [42, 43, 63] are last value, stride, and context-based predictors. They are based on the history pattern seen by individual instruction operands. Nakra et al. [52] proposed *path-based value predictors* to predict values along different control-flow paths. The idea of correlating the prediction history with control-flow traces is employed in multithreaded architectures [47, 48, 60]. These architectures (as listed) achieve significant performance improvement by limiting the speculation to only high-confidence, live-in registers.

Register dependence speculation is performed in conjunction with register communication. It speculates whether a register is written for the last time in a thread. After the predicted point, the register communication hardware (or software) assumes that there is no further read-after-write dependence, caused by this register, from this thread to the others. A register forwarding strategy, *Spec_send*, proposed by Vijaykumar [72] speculatively forwards a register when it is unlikely to be further updated. An update probability is assigned by the compiler to each register in each basic block of a task, using profile information and data flow analysis.

UPC Speculative Multithreaded predicts the number of writes to each register by

each thread. Once a thread performs the predicted number of writes, it will forward the register to the next thread. Misprediction is detected when the number of actual writes exceeds the predicted number.

2.6.3 Memory Speculation

Each thread processing unit (or processor) is typically equipped with a private memory buffer or L1 cache to keep results from the thread execution. In the sequential control-flow, *RAW* or *read-after-write* dependence occurs when an instruction reads a value which has been written by its predecessor; *WAR* or *write-after-read* dependence occurs when an instruction writes a new value to a memory location (or register) after the old one has been read by its predecessor; and *WAW* or *write-after-write* dependence occurs when an instruction writes a value to the same memory location (or register) as its predecessor. In the multithreaded execution, the multiple versions of memory data must be handled properly to honour the RAW, WAR, and WAW dependencies. Generally, a load must see the latest store to the same address (RAW rule) and should not be aware of stores to the same location by successor threads (WAR rule). It must be squashed and re-executed if it has read the wrong version of the data. Finally, concurrent threads perform write-back to the shared memory in the correct sequential order (WAW rule).

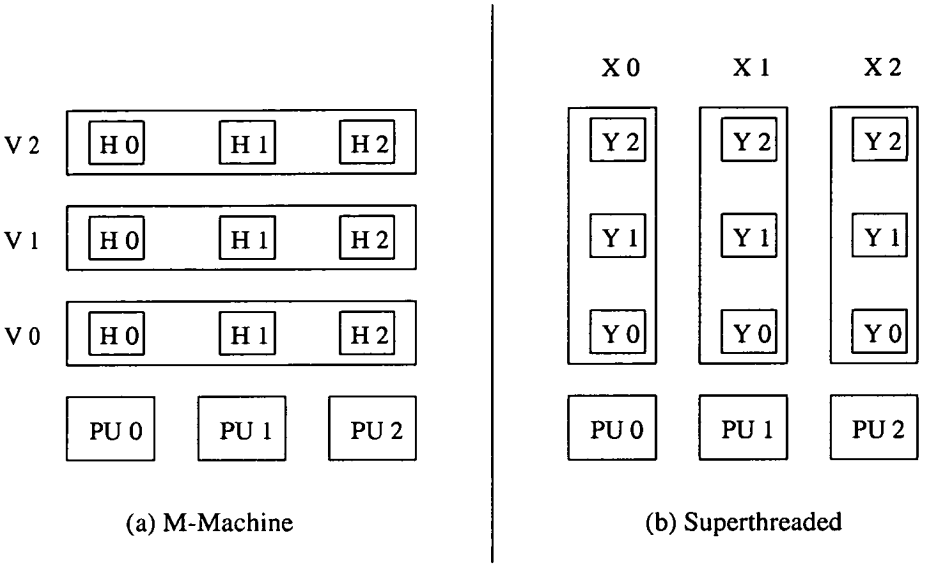
Hydra and STAMPede allow threads to dynamically switch between speculative and non-speculative execution. A speculative region is marked by *start_speculation* and *end_speculation* instructions. Because a thread can store to the shared memory when it is non-speculative, the compilers must ensure that store operations outside the speculative regions are safe. They use hardware to detect dependence violation and software to control recovery actions. Hydra relies on a snooping-bus-based mecha-

nism. When a processor writes back to the next level shared memory (L2 cache), all the other processors watch the write bus to detect the violation. On the other hand, STAMPede extends invalidation-based cache coherence. When an epoch stores to a location that has been speculatively loaded, it sends invalidation signals to the consumer epochs. The consumers detect the violation by comparing their sequence orders with the producer's.

More complicated approaches include *Address Resolution Buffer (ARB)* [26, 27] and *Speculative Versioning Cache (SVC)* [30]. Both of them aggressively perform memory speculation, i.e. every load and store can be executed as soon as its address is known even if memory references in the preceeding tasks are still unresolved. ARB is a centralised structure. It keeps all versions of the data from all tasks, and consequently suffers from limited bandwidth and long access delays. In contrast, SVC is a decentralised structure. The memory references are spread across multiple caches. Although it solves the problems in the ARB, the SVC incurs lower hit rate and larger amount of communication between caches.

2.7 Hierarchical Organisation and Clusters

The *M-Machine* [23] has two levels of concurrency. As illustrated in Figure 2.1(a), *V-threads* share the same set of processing units and can be swapped in and out of the processors. A V-thread is composed of subthreads or *H-threads* which simultaneously execute on separate processing units. In contrast, the two-dimensional Superthreaded [68], as shown in Figure 2.1(b), has *X-threads* allocated to different processing units, each of which comprises of multiple resident *Y-threads*. Normal policies for context switching are round-robin and event-trigger (e.g. cache misses). A major advantage



(a) M-Machine (b) Superthreaded

Figure 2.1: Two-level multithreaded models

of hierarchy in the M-Machine and the two-dimensional Superthreaded is in its ability to exploit more parallelism, by hiding the long latency of operations such as memory access and inter-thread communication.

Zahran and Franklin [78] have proposed *Hierarchical Multithreading (HMT)*. Their architecture is basically a network of Multiscalar processors. A program is partitioned into *supertasks* which are assigned to the Multiscalar nodes. The supertasks are further broken into tasks and assigned to processing units within the nodes. The HMT takes advantage of coarse-grained thread-level parallelism since the supertasks are typically far apart in the sequential control-flow order. In addition, control and data dependencies between them are minimised in order to limit the amount of communication between the Multiscalar nodes.

Simultaneous Subordinate Microthreading (SSMT) [15] employs a concept similar to interrupt handling. Events, such as branch mispredictions and cache misses, occurring as a result of a (primary) thread's execution automatically spark specialist

microthreads. The microthreads execute optimisation routines which are written in the internal machine format and stored on-chip. During the microthread initialisation, these routines are loaded into the decode/rename stage and issued simultaneously with the primary thread's instructions. Another example of using separate threads to handle exceptions is described in Zilles et al. [80]. Exception threads are sparked to fetch and execute exception handlers before the normal execution resumes. By fetching the exception handlers separately, the main threads need not squash the instructions following the ones that cause the exceptions, and are able to execute the independent ones in parallel with the exception handling.

In Dorai and Yeung [17], *foreground* threads perform high-priority or critical computation whereas *background* threads perform low-priority ones. They aimed at making the background threads *transparent* or having almost no impact on the performance of the foreground threads. Hardware resource are divided into three classes: instruction slots, instruction buffers, and memories. Competition for each type of resources affects the performance of the foreground threads differently. For example, the foreground threads are disrupted for only single cycle if they lose out on instruction slots such as fetch and functional units. However, they may be disrupted for several cycles if they lose out on instruction buffers. Although there is little contention for memory resources such as caches and branch prediction tables, interfering accesses by the background threads may cause performance degradation in the foreground threads.

As an architecture is scaled up, the wire delays become a hurdle to the overall performance. Because of this, there have been proposals to group multiple processing units into clusters [21, 38, 39, 47]. Programs are typically partitioned, either statically or dynamically, to exploit *communication locality*. In general, threads that cause frequent communication are allocated to processing units in the same clusters.

2.8 Other Techniques

A new dynamic resource allocation approach has been introduced in α-*Coral* architecture [77]. It has a large register file and a program counter queue holding the states of all currently-active threads in the processor, both of which are centralised. New threads can be spawned until the program counter queue is full. Upon thread initialisation, a segment of the shared register file is allocated to the thread. The size of the segment depends on the number of registers each thread requires, allowing flexibility in the resource management. However, a drawback of the centralised structures is poor scalability.

2.9 Chapter Summary

This chapter has investigated some of the fundamental issues in multithreaded execution. These include the creation, initialisation, and retirement of threads; the interaction between concurrent threads including communication, synchronisation, and thread-level speculation; and hierarchical structures. Relevant software and hardware techniques were reviewed. Some of these have inspired our compiler and architecture designs in the forthcoming chapters.

Chapter 3

The Multithreaded Processor Architecture and The Compiler

The target architecture is a CMP-based multithreaded processor which was inspired by hardware simplicity of the Superthreaded model [70]. The initial design was presented in [5, 6, 34]. First, hierarchical multithreaded execution is described briefly in Section 3.1, followed by the architectural details in Section 3.2 which include novel features to support hierarchical execution, register synchronisation and forwarding, and speculation. The multithreaded instructions are described in Section 3.3, and the implementation of the multithreaded processor simulator in Section 3.4. Finally, the multithreaded compiler is described in Section 3.5.

3.1 Hierarchical Multithreaded Execution

In part of the master/slave execution model [18, 23, 68, 78], a thread can execute a command to form a cluster of slave TPUs during program execution. Each slave thread, which runs on the slave TPU, could in turn form a cluster at the next level,

and so on recursively. The master thread could free its slave TPUs by executing a command to release the cluster. Hence, clusters in our context are dynamic and logical entities. The thread processing units in a cluster are logically connected to each other in a uni-directional ring and operate in the predecessor/successor style [45, 65, 68].

Threads can be created or forked in two directions: the master thread forks a new slave in the *vertical* direction, while the slave thread forks the next one in the *horizontal* direction. When a thread forks a new thread, it becomes the parent of that new thread. If a master thread T_0 vertically forks a slave thread T_1, and T_1 horizontally forks another slave thread T_2, then the relationships between T_0, T_1, and T_2 would be:

- For master/slave relationships, T_0 is the master of T_1 and T_2 (conversely, T_1 and T_2 are the slaves of T_0).
- For parent/child relationships, T_0 is the parent of T_1, and T_1 is the parent of T_2. Thus, T_1 is the child of T_0, and T_2 is the child of T_1, respectively.

As in the predecessor/successor model, the slaves retire and update the cluster's state, instead of the processor's state, in a sequential order. Since the cluster's state is maintained by the master thread, this is also equivalent to *merger* in the master/slave model. Upon merger, register values, program counter, and speculative results of the slaves are transferred to the master's.

In order to incorporate this idea into the original design [5, 6, 34], additional features were introduced in the *Global Thread Control Unit (GTCU)*, *Local Thread Control Units (LTCUs)* and *Speculative Buffers*. Furthermore, additional multithreaded instructions were proposed to support hierarchical and speculative execution.

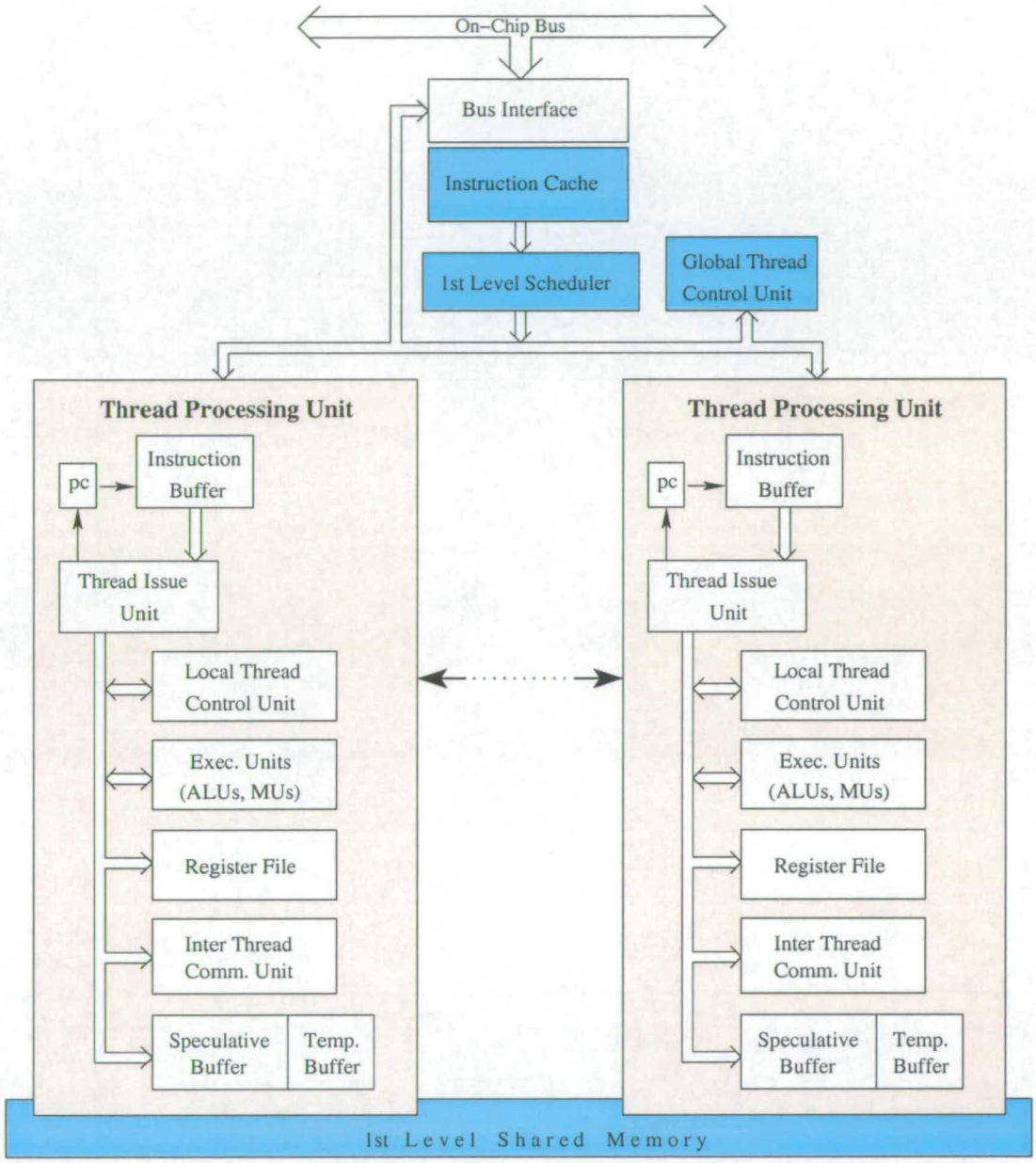


Figure 3.1: The target multithreaded architecture

3.2 Description of the Architecture

Figure 3.1 depicts an overview of the multithreaded architecture. The processor consists of a number of identical *Thread Processing Units (TPUs)*. At the start of program execution, the *First Level Scheduler (FLS)* fetches instructions from the central instruction cache and passes them to the instruction buffer of a head thread which, by default, always runs on TPU 0.

3.2.1 Global Thread Control Unit (GTCU)

As the architecture relies on static program partitioning, thread sequence according to the sequential semantics has to be conveyed from the compiler to the hardware. Besides controlling the retirement order of concurrent threads, the sequence information is needed for handling multiple versions of loads/stores in speculative execution. The *Global Thread Control Unit (GTCU)* was added to the original design, which maintains the relative order, by ascending sequence numbers, of all the active threads in the processor. If multiple threads have the same sequence number, then they are ordered by the time of creation, starting from the oldest. A sequence number is assigned to a thread either explicitly or implicitly, as described next.

1. Explicit assignment. For a normal fork operation (*frk* instruction in Section 3.3.1), a sequence number is given as an argument of the fork.

2. Implicit assignment. In the cases of vertical and horizontal fork operations (*yfrk* and *xfrk* instructions in Section 3.3.2), a child is given the same sequence number as the parent's, and the master/slave relationship has priority over the parent/child relationship. Thus, slave threads are inserted in the order list immediately after the master and following the parent/child relationship between them.

The thread sequence is updated each time a new thread is forked or an existing thread retires, by receiving signals from the Local Thread Control Units (LTCUs). The GTCU also maintains a pointer to the head thread, which is generally the oldest running thread on the processor.

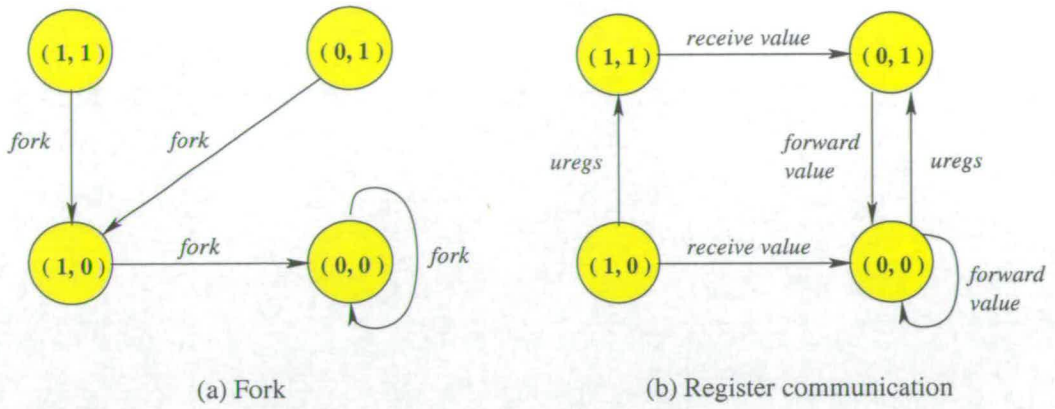
3.2.2 Thread Issue Unit (TIU)

The TIU decodes instructions and passes them to the corresponding execution units. Instructions are issued in-order from the instruction buffer but can be executed out-of-order as soon as resources are available. The instruction-level parallelism is exposed by the compiler's instruction scheduling and optimisation techniques. Normal arithmetic and memory instructions are sent to ALUs and MUs respectively, while multithreaded instructions are sent to the LTCU.

3.2.3 Local Thread Control Unit (LTCU)

The LTCU executes the multithreaded instructions. It also maintains the following information:

- *Parent Address.* It is set when a thread is initialised.
- *Child Addresses Table.* A child address is added to the table if the fork operation succeeds. As soon as the parent thread retires, its children will be notified to invalidate the parent address.
- *Slave Addresses Table.* This table is set when a thread successfully forms a slave cluster and is cleared when the cluster is released.
- *Master Address.* It is set for a slave thread to retain the address of its master.



(a) Fork (b) Register communication

Figure 3.2: State transitions for (W, U) in a register

3.2.4 Register File

A simple register synchronisation and forwarding mechanism was proposed. As a preliminary study, the register forwarding is restricted to only from parent to child threads. Each register in the local register file is associated with the following 2 bits:

- *W bit.* This bit is set for the child thread's register to enforce synchronisation until the register is forwarded from its parent. If the thread tries to read a register whose W bit is set, then it has to *wait* until the bit is turned off. If the thread writes this register for the first time (before any read), then the W bit is turned off since it no longer has to wait for the value forwarded from the parent.

- *U bit.* This bit is set, prior to forking a new thread, for the parent thread's register whose value is *unavailable* to the child.

When a new thread is initialised, the register values are copied from the parent's register file to the child's. Figure 3.2 (a) depicts the state transitions of a register from parent to child: $(W(parent), U(parent)) \xrightarrow{fork} (W(child), U(child))$. The initial state

of the child's register is set to either (1,0) or (0,0). The former implies that the register might be live-in; the latter implies that the register is dead-in. If the state of the parent's register is (1,1) or (0,1), then the child's state is set to (1,0) which indicates that the register might be live-out from the parent but its value is not yet available to the child.

After the parent produces a value for the live-out register, it can be forwarded to the child. The forwarding operation resets the U and W bits in the parent's and child's registers, respectively. Figure 3.2 (b) depicts the state transitions of a register of the same thread due to the register communication: $(W,U) \xrightarrow{action} (W,U)$. The thread can forward a register to its children only if the state is (0,0) or (0,1), i.e. it is not waiting for that register itself. Upon receiving the values, the receivers set their corresponding W bits to 0. Finally, a set of live-out registers whose values have not yet been produced can be declared by executing *uregs* (see Section 3.3). This instruction sets the specified U bits to 1 and consequently enforces synchronisation in the successor threads when they try to read those registers.

3.2.5 Speculative Buffer

The speculation is almost entirely controlled by the compiler. The hardware support is very simple, as described next.

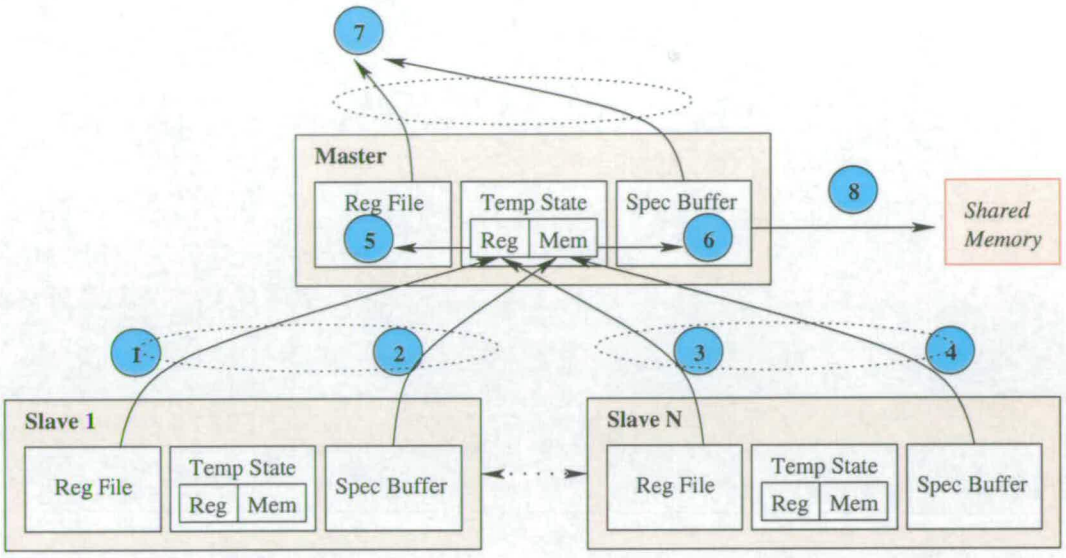
A thread can switch between non-speculative and speculative modes during its execution, in the same style as in STAMPede [65]. When the thread becomes speculative, it writes to the speculative buffer instead of to the shared memory. These stores are flushed to the memory when the thread commits. If the thread stops without committing these stores, then the speculative buffer is simply cleared. For a load operation, the thread should see the latest version of the data as if the program is executed in the sequential order. Firstly, it checks the load address in its own buffer. If the address

is not found, it will lookup the predecessors' buffers. Finally, if the address is still not found in any of the predecessors' buffers, then it will load from the memory. The information as to which threads are the predecessors of the current thread is obtained by scanning the thread order list maintained by the GTCU. If the dependency distance between threads is large, i.e. a thread is data dependent on a predecessor which is far ahead of itself in the order list, then the overhead of loading can be quite high. Compiler techniques such as loop unrolling [20] can reduce the dependency distance so that the thread is only data dependent on its immediate predecessor.

However, both misspeculation detection and recovery are performed in the software. The misspeculation is handled by aborting the wrong thread and starting a new one to execute the correct path.

In the non-speculative mode, the thread directly reads from and writes to the shared memory. The compiler determines whether a load/store operation is safe and chooses the execution mode accordingly, in order to guarantee the program correctness. For example, a thread can store its result in the speculative buffer and then load data from the shared memory, by switching from the speculative mode (before the store) to the non-speculative one (before the load).

In the hierarchical execution, if the master thread is speculative, then its slaves should also run in the speculative mode. Our execution model expects the master to only fork slave threads to execute parts of the program which are logically ahead of itself. As shown in Figure 3.3, there is a temporary storage inside the speculative buffer, which maintains the cluster's state. When the slaves are merged into the master, their register and memory updates are collected as the master's temporary state (or the cluster's state). As soon as the clustered execution is completed and the cluster is freed, the temporary register values (including the program counter) will be transfered to the



Token	Description
1, 2	Slave 1st is merged. Registers and speculative stores are saved to the master's temporary state.
3, 4	Slave Nth is merged. Registers and speculative stores are saved to the master's temporary state.
5, 6	Master releases cluster. 5 : Temporary register updates are transfered to register file. 6 : Temporary speculative stores are transfered to speculative stores.
7	Master retires. Registers and speculative stores are saved to temporary state of the higher-level master.
8	The highest-level master commits speculative stores to the shared memory.

Figure 3.3: Retirement actions in the hierarchical-speculative execution.

current register values while the speculative stores from the slaves will be transfered to the master's speculative stores instead of being flushed to the shared memory.

3.2.6 Inter-thread Communication Unit

The inter-thread communication unit takes care of the signal transmission between the TPUs. It contains a signal buffer and, depending on the implementation, signal handlers for some particular signals. In the absence of the signal handlers, the signal

transmission can be used as a synchronisation mechanism, for example, between the memory load/store operations.

When a signal is transmitted, its number is written in the buffer of the target TPU. The thread that executes *wait signal* instruction checks its signal buffer and blocks until the signal has arrived. Then it either invokes the signal handling routine or simply continues its execution, after which the signal is removed from the buffer. A signal might be lost before it has been processed should the buffer be full and the old signal be replaced by a new one.

3.3 Multithreaded Instructions

A subset of the standard MIPS instructions [22, 37] was augmented with multithreaded instructions. These can be categorised into 4 groups: basic instructions (Tables 3.2 and 3.3), auxiliary instructions (Table 3.4), instructions that support hierarchical execution, (Tables 3.5 and 3.6), and instructions that support speculative execution (Table 3.7). In the tables, *$s* and *$d* denote source and destination register operands, respectively; *L* is a label; and *I* is an integer value. Description and pseudo-code of each instruction are also provided in the tables. Pseudo-functions, other than the ones associated with the instructions, perform operations as indicated by the names of these functions (examples are listed in Table 3.1).

Most multithreaded operations are *guarded.* Semantics of a guarded operation is to evaluate the guard operand: if the condition is true, then the instruction is executed; otherwise it is treated as a *nop* instruction. Similar to the thread creation or forking in SPSM [18], the fork-operation may succeed or fail depending on the availability of resources at run-time. Subsequent multithreaded instructions must therefore be exe-

cuted under guarded conditions to preserve the program correctness, particularly those involved in synchronisation and communication as they could potentially cause deadlocks. In order to prevent premature program termination, a thread needs to check on occasion if it has successfully forked the next one before it retires. Thus, the retirement operation is also guarded.

3.3.1 Multithreaded Instructions Group 1

Tables 3.2 and 3.3 summarise the basic instructions for simple non-speculative multithreaded programs. They are modified from the preliminary proposal in [5, 34].

Table 3.1 Examples of pseudo-functions

	Function
1	`TPU& get_TPUs(num_TPUs)`
	`// Get available TPUs and return a pointer to these TPUs`
2	`bool TPU::avail()`
	`// Check whether the TPU is available`
3	`void thread_init(TPU&, sequence, label)`
	`// Initialise a new thread on the TPU`
4	`void cluster_init(master_thread, TPU&)`
	`// Initialise a new cluster`
	`// Thread's operation`
5	`void thread::wait_signal(signal)`
6	`void thread::get_signal(signal)`
7	`void thread::save_pc(label)` `// Set OPC = PC and PC = label.PC`
8	`void thread::restore_pc()` `// Set PC = OPC`
9	`void thread::interrupted()`
10	`void thread::stop(merge)`
11	`void thread::commit()`
12	`void thread::set_head(thread)` `// Nominate a new head thread`
	`// Master thread's operation`
13	`void master_thread::pass_signal(slave_thread&, signal)`
14	`void master_thread::cluster_release()`
15	`void master_thread::cluster_abort()`

Table 3.2 Multithreaded Instructions Group 1 (continued in Table 3.3)

Instruction	Description

frk *$d, $s1, L*	Fork a new thread to execute target label *L*. Return *d = TRUE*, if successful. *s1* is a sequence number associated with the new thread.
	`op frk(op s1, op L) {`
	`  if (pt_TPUs = get_TPUs(1)) {`
	`  d = TRUE;`
	`  thread_init(pt_TPUs, s1, L);`
	`  }`
	`  else d = FALSE;`
	`  return d;`
	`}`

stp *$s1, $s2*	If guard *s1* is set, then stop. If it is the head thread, set thread *s2* as the new head.
	`void stp(op s1, op s2) {`
	`  if (s1) {`
	`  thisthread.set_head(s2);`
	`  thisthread.stop(merge = FALSE);`
	`  }`
	`}`

sstp *$s1, $s2*	Wait for synchronisation signal and pass it to thread *s2*. If guard *s1* is set, stop. If it is the head thread, set *s2* as the new head.
	`void sstp(op s1, op s2) {`
	`  thisthread.wait_signal(SYNCH);`
	`  if (s1) {`
	`  thisthread.set_head(s2);`
	`  thisthread.stop(merge = FALSE);`
	`  }`
	`  s2.get_signal(SYNCH);`
	`}`

Table 3.3 Multithreaded Instructions Group 1 (continued from Table 3.2)

Instruction	Description

psg *$s1, $s2, I*	If guard *s1* is set, pass signal *I* to thread *s2*.
	`void psg(op s1, op s2, op I)`
	`{ if (s1) s2.get_signal(I); }`

wat *$s1, I*	If guard *s1* is set, wait until signal *I* is received.
	`void wat(op s1, op I) {`
	`if (s1) {`
	`thisthread.wait_signal(I);`
	`signals[I].handler(thisthread);`
	`}`
	`}`

isg *$s1, $s2, L*	If guard *s1* is set, interrupt the execution of thread *s2*. The interrupted thread jumps to label *L*.
	`void isg(op s1, op s2, op L) {`
	`if (s1) {`
	`s2.save_pc(L);`
	`s2.interrupted();`
	`}`
	`}`

mop *$s1*	If guard *s1* is set, move the old program counter to the current program counter.
	`void mop(op s1)`
	`{ if (s1) thisthread.restore_pc(); }`

The **frk** instruction forks a new thread on an available TPU and returns *TRUE*, or returns*FALSE* if no TPU is available. The TPU address of the newly-forked thread is retrieved by executing **cadr** (see Table 3.4). An alternative design is to return the child's TPU address if the fork succeeds, or an *INVALID* value (e.g. -1) if it fails. However, we opted for the first approach because the values *TRUE/FALSE* are handy to use either as guards in the subsequent multithreaded instructions or as operands in conventional branch instructions (e.g. *beqz*).

A thread can stop either with or without waiting for a synchronisation signal, by executing either the **stp** or **sstp** instructions. Before the thread stops, it will nominate a new head thread. The nomination is valid only if it is currently pointed to by the *head* pointer. If the nominated thread is not active, then the GTCU will move the *head* pointer to the next thread in the order list.

The **psg** and **wat** instructions communicate signals. The **psg** is a non-blocking send while the **wat** is a blocking receive. The **psg** puts the signal number in the receiver's signal buffer. When the thread executes **wat** and receives the signal, it either performs a sequence of actions as specified by the signal handler or continues its execution if there is no handler for that signal.

The **isg** instruction interrupts the execution of the target thread. Having been interrupted, the thread saves its current program counter (PC) to the old program counter (OPC) before branching to the interrupt handling routine. The thread may invoke a default hardware procedure by sending a signal to itself and receiving that signal. Finally, the **mop** instruction can be inserted at the end of the interrupt. When the thread reaches **mop**, it copies the content of the OPC back to the PC and resumes its execution.

Table 3.4 Multithreaded Instructions Group 2

Instruction	Description

adr *$d*	Return the address of this thread.
	`op adr()`
	`{ return thisthread.address; }`

padr *$d*	Return the address of the parent of this thread.
	`op padr()`
	`{ return thisthread.parent.address; }`

cadr *$d*	Return the address of the most recent child of this thread.
	`op cadr()`
	`{ return thisthread.children[last].address; }`

hadr *$d*	Return the address of the head thread.
	`op hadr()`
	`{ return thisthread.head.address; }`

3.3.2 Multithreaded Instructions Group 2

The auxiliary instructions such as **adr**, **padr**, **cadr**, and **hadr** are summarised in Table 3.4. They do not have guard operands since the execution of these instructions have no side-effect on the state of the TPU or the processor. The **cadr**, as mentioned earlier, is used as a complement to **frk** and might be omitted in an alternative design. The **adr**, **padr**, and **hadr** can be replaced by the use of the additional software routines to keep track of the thread information, as was done in the preliminary work [34].

3.3.3 Multithreaded Instructions Group 3

The instructions in Tables 3.5 and 3.6 support hierarchical multithreaded execution. They can be emulated by a sequence of the basic multithreaded instructions described earlier, at the expense of additional software thread manipulation costs.

Table 3.5 Multithreaded Instructions Group 3 (continued in Table 3.6)

Instruction	Description

cform *$d, $s1*	Form a cluster of *s1* slave TPUs. Return *d* = *TRUE*, if successful.
	`op cform(op s1) {`
	`  if (pt_TPUs = get_TPUs(s1)) {`
	`  d = TRUE;`
	`  cluster_init(thisthread, pt_TPUs);`
	`  }`
	`  else d = FALSE;`
	`  return d;`
	`}`

yfrk *$s1, $d, L*	Vertical fork. If guard *s1* is set, fork a new thread on the first slave TPU, and return *d* = *TRUE*, if successful. The new thread executes label *L*.
	`op yfrk(op s1, op L) {`
	`  if (s1 && pt_TPUs =`
	`  thisthread.slave_TPUs[0].avail()) {`
	`  d = TRUE;`
	`  thread_init(pt_TPUs, thisthread.seq, L);`
	`  }`
	`  else d = FALSE;`
	`  return d;`
	`}`

xfrk *$s1, $d, L*	Horizontal fork. If guard *s1* is set, fork a new thread on the next slave TPU, and return *d* = *TRUE*, if successful. The new thread executes label *L*.
	`op xfrk(op s1, op L) {`
	`  if (s1 && pt_TPUs =`
	`  thisthread.next_TPU.avail()) {`
	`  d = TRUE;`
	`  thread_init(pt_TPUs, thisthread.seq, L);`
	`  }`
	`  else d = FALSE;`
	`  return d;`
	`}`

Table 3.6 Multithreaded Instructions Group 3 (continued from Table 3.5)

Instruction	Description

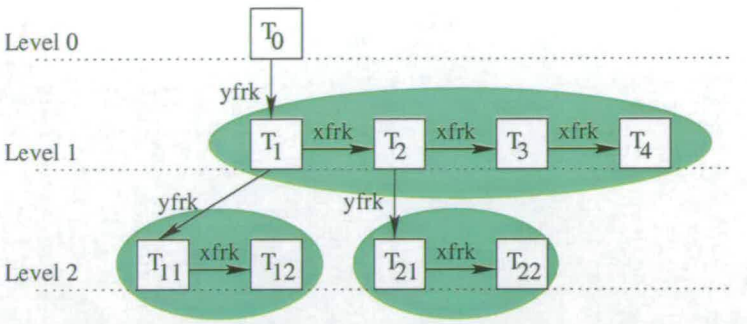
crels *$s1* *$s2*	Cluster release. Execute if guard *s1* is set. If *s2* = *TRUE*, send synchronisation signal to slaves and free the cluster when the signal returns. Otherwise, abort the slaves and release the cluster.
	`void crels(op s1, op s2)`
	`{`
	`  if (s1) {`
	`  if (s2) {`
	`  thisthread.pass_signal(pt_slaves, SYNCH);`
	`  thisthread.cluster_release();`
	`  }`
	`  else thisthread.cluster_abort();`
	`  }`
	`}`

xstp *$s1*, *$s2*	Similar to sstp. Synchronisation signal is passed to the next slave or back to the master if it is the last active slave. *s2* indicates whether the slave's state is merged into the master's.
	`void xstp(op s1, op s2) {`
	`  thisthread.wait_signal(SYNCH);`
	`  if (s1) thisthread.stop(merge = s2);`
	`  thisthread.pt_next.get_signal(SYNCH);`
	`}`

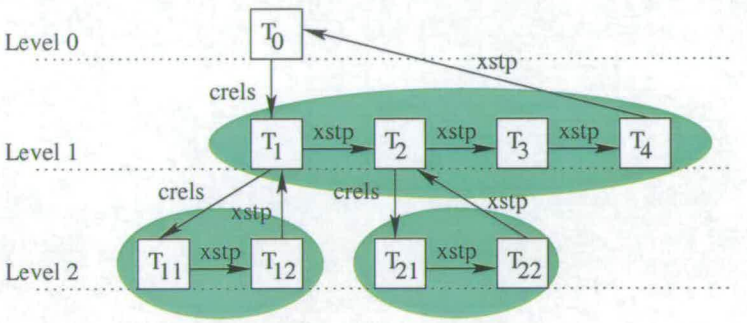
The **cform** instruction checks for available TPUs in the processor and reserves them as slave TPUs. A TPU is considered to be available if it is unclustered and there is no thread running on it.

A thread can fork new threads in 2 directions: vertical fork (**yfrk**) and horizontal fork (**xfrk**). The **yfrk** is executed by the master thread to fork a child on the first slave TPU. The slave then executes **xfrk** to fork a successor thread on the next slave TPU. The slaves are inserted in the list maintained by the GTCU, after their master and in the order in which they are forked. Hence the sequence number is not given in both instructions. An assumption is that *the master should only fork the slaves to execute program partitions which are encountered after the one executed by the master, according to the sequential semantics*. Figure 3.4 (a) demonstrates the use of **yfrk** and **xfrk** where the threads in level 1 execute the outer loop iterations and the ones in level 2 execute the inner loop iterations. The order of the current running threads is depth-first or in-order: T_0, T_1, T_{11}, T_{12}, T_2, T_{21}, T_{22}, T_3, T_4.

The master and slave threads synchronise by executing **crels** and **xstp** instructions, respectively. This is equivalent to *merging* in the SPSM model [18]. Unlike the SPSM, where the main thread's merging point is implicitly the starting address of the future thread, the master thread in the hierarchical model explicitly executes **crels**. It passes the synchronisation signal to the first active slave and waits until all the slaves have been retired. When the slave executes **xstp**, it waits for the synchronisation signal before merging its state into the master's temporary state. If it is the last active slave in the cluster, then it passes the signal back to the master. Otherwise, it passes the signal to the next slave. When the signal returns to the master, it frees the slave TPUs and transfers its temporary state to the current state. The execution resumes after the last instruction executed by the last slave (as if the slaves' execution has been sequentially



(a) Hierarchical fork



(b) Hierarchical synchronisation

Figure 3.4: Hierarchical multithreaded execution

performed by the master itself). If the slave cluster is aborted, then all the slaves are interrupted from their execution and stop.

Figure 3.4 (b) demonstrates the synchronisation between the threads in Figure 3.4 (a). The order of execution is the in-order traversal of the tree: $T_0 \rightarrow T_1 \rightarrow T_{11} \rightarrow T_{12} \rightarrow T_2 \rightarrow T_{21} \rightarrow T_{22} \rightarrow T_3 \rightarrow T_4$. Since T_{11} and T_{12} execute the inner loop of the first outer loop iteration (executed by T_1), then they must be merged into T_1 before the synchronisation signal is passed to the next outer loop iteration (executed by T_2).

When T_0 aborts its cluster $\{T_1, T_2, T_3, T_4\}$, T_1 and T_2 also abort their next-level clusters before stopping.

3.3.4 Multithreaded Instructions Group 4

The instructions in the final group, as shown in Table 3.7, support speculative execution. The **safe** instruction switches between the non-speculative and the speculative modes. By default, a thread is non-speculative when it starts the execution. When it becomes speculative, all the store operations write to the speculative buffer instead of to the shared memory.

Because the speculative buffer is cleared when the thread stops, it must explicitly execute **cmmt** to write to the memory if the speculation is correct. In the case of misspeculation, the guard operand of the **cmmt** can be set to *FALSE*. The thread will simply stop without committing the results from the speculation.

The **uregs** and **fregs** instructions manipulate the U and W bits of the thread's registers. The registers whose corresponding bits are to be set are specified by the mask which encodes base registers 0-31 and the offset which is an integer to be multiplied by 32, $(\textit{register number} = \textit{base register number} + 32(\textit{offset}))$. The **uregs** sets U bits to *TRUE* which indicates that the corresponding registers are unavailable to the successor threads. The **fregs** instruction sets U bits to *FALSE* and forwards the register values to the thread's successors. Upon receiving the values, the successors will set their W bits to *FALSE*. If the thread executes **fregs** when it has no children, then the specified U bits are simply switched off.

Table 3.7 Multithreaded Instructions Group 4

Instruction	Description

safe *$s1, $s2*	Execute if guard *s1* is set. If *s2* = *TRUE*, the following stores will write-through to memory. Otherwise, the following stores will write to speculative buffer.
	`void safe(op s1, op s2)`
	`{ if (s1) thisthread.set_mem_access(s2); }`

cmmt *$s1*	If guard *s1* is set, wait for synchronisation signal and commit speculative stores to memory.
	`void cmmt(op s1) {`
	`if (s1) {`
	`thisthread.wait_signal(SYNCH);`
	`thisthread.commit();`
	`}`
	`}`

uregs *$s1, I ($s2)*	If guard *s1* is set, set *U* bits of the registers specified by mask *I* and offset *s2* to *TRUE*.
	`void uregs(op s1, op I, op s2) {`
	`if (s1) {`
	`while (r = decode(I, s2))`
	`thisthread.regs[r].set_U(TRUE);`
	`}`
	`}`

fregs *$s1, I ($s2)*	If guard *s1* is set, forward the registers specified by mask *I* and offset *s2* to the child threads.
	`void fregs(op s1, op I, op s2) {`
	`if (s1) {`
	`while (r = decode(I, s2)) {`
	`thisthread.regs[id].set_U(FALSE);`
	`for (i=0; i<=last; i++)`
	`thisthread.children[i].get_reg(r);`
	`}`
	`}`
	`}`

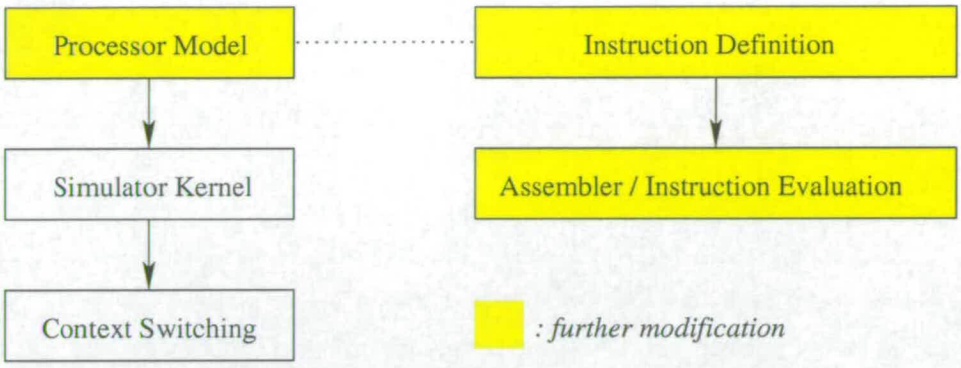


Figure 3.5: An overview of the simulator

3.4 The Multithreaded Processor Simulator

A sequential processor simulator [55] was modified to handle the multithreaded processor architecture. The basic multithreaded features were implemented in [5]. It was enhanced considerably to reflect the details described in Sections 3.1, 3.2, and 3.3.

3.4.1 Simulator Framework

The framework is based on a process-based, discrete-event simulator. Figure 3.5 depicts an overview of the simulator. It was implemented in C++ and can be divided into three layers: processor model, simulator kernel, and context switching.

3.4.1.1 Simulator Kernel

The simulator kernel is a general-purpose library for discrete-event simulation. It defines class *entity*, from which the processor components are derived. Entities are divided into two types: (1) participating entities such as First Level Scheduler (FLS), Thread Processing Units (TPUs), and Arithmetic and Logic Units (ALUs); and (2)

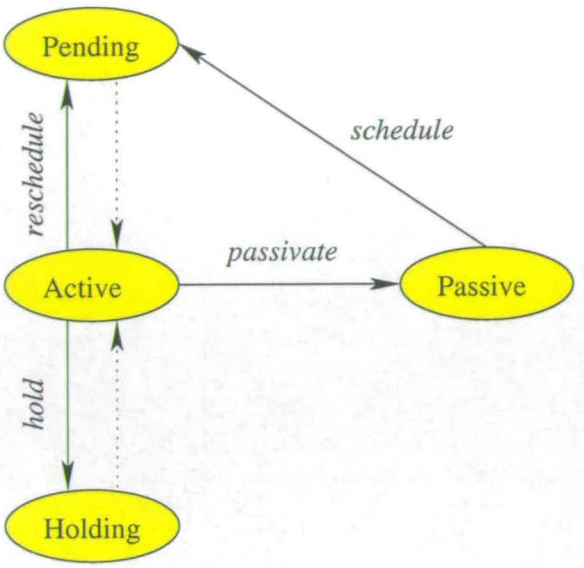


Figure 3.6: State transitions of a participating entity

non-participating entities such as register files.

A participating entity may be in one of four states: passive, active, holding, or pending. Figure 3.6 depicts the transitions between these states. Solid lines denote explicit transitions made by function calls and dotted lines are implicit transitions made by the simulator kernel. Passive entities have no control over the progress of the simulation. They can be scheduled and placed in the pending queue (see below) by active entities. The active entities may change the state of the simulation, schedule passive entities, or reschedule themselves. Holding entities are ones waiting for the state of the simulation to meet certain conditions before activating. Pending entities are ones being scheduled to activate after certain times; they are held in the pending queue which is ordered by the activation time. Both holding and pending entities are passive. Once the current active entity deactivates and there is no other active entity, the simulator kernel searches for a new active entity. It first checks in the holding list. If no holding

entity is able to activate in the current state of the simulation, the kernel picks the first entity in the pending queue and advances the simulation time.

3.4.1.2 Context Switching

The context-switching layer is a layer on which the rest of the simulator is built. Entities are derived from the base class *context*. Each context is an operating system thread. Mechanisms to maintain and switch between contexts are implemented in this layer. It is also the only layer that handles operating system functions.

3.4.1.3 Processor Model

The processor is modelled in the top layer. Its components are derived from the base class *entity* and the behaviours of these components are implemented using state transition functions. For example, the Thread Issue Unit (TIU) inside an active Thread Processing Unit (TPU) repeatedly performs the following operations:

1. Get a new instruction.
2. If there is a buffer hit,
   (a) Hold until the registers required are unlocked.
   (b) Fetch the source registers and lock the destination register.
   (c) Issue the instruction to (i.e. schedule) Arithmetic and Logic Unit, Memory Unit, or Local Thread Control Unit.
   (d) Reschedule itself, accounting for issue cycle time.
3. If there is a buffer miss,
   (a) Send a request to First Level Scheduler and passivate.

However, components such as the *buses* and the *bus interface* are not simulated separately. It was assumed that the delay for a processor component to perform its task has included bus delay if that component accesses the buses, and bus contention is lumped in the contention for that component. In practice, the bus delay is also affected by distances between the components. For example, in a large application where several clusters of TPUs are allocated to execute several program partitions, the TPUs in large clusters may be more scattered than in smaller ones. Our multithreaded execution models, which will be described in Chapters 4 and 5, permit communication between parent and child threads only. Assuming that they often (if not always) execute on neighbouring TPUs, the communication delay which includes the bus delay is set to be uniform.

Delays are measured in terms of number of *time units*. The absolute number is in itself less important, but the ratio between time delays should be realistic or correspond to the architectural assumptions being made - the actual execution time can be correctly estimated once a clock speed has been set. For example, a normal ALU instruction is split into four operations: fetch instruction, read registers, execute, and write back. Thus, the delay is set to 4 time units. On the other hand, the delays of the auxiliary multithreaded instructions (Section 3.3.2) are expected to be short, as these instructions are frequently used to support the multithreaded execution. The delay of the First Level Scheduler (FLS) is proportional to its fetch bandwidth and the size of the instruction buffer in the TPU. For instance, if the fetch bandwidth is 2 instructions per time unit and the TPU's buffer holds 10 instructions, then the FLS delay for processing a request from a TPU would be 5 time units.

The Global Thread Control Unit (GTCU) was implemented as a non-participating entity in the simulator because it does not perform any action other than maintaining

threads' information. Accesses to the GTCU are managed in a multiple-readers/single-writer style as the thread sequence can be updated by only one thread at any time (experiments on the GTCU's access delay are reported in Appendix B).

3.4.1.4 Instruction Definitions

The instruction set module is defined separately from the processor model. An *instruction definition* is implemented for each opcode and has two main functions: syntactic analysis and instruction evaluation. At the start of the simulation, an input file is parsed. For each instruction, syntactic analysis is performed, which includes checking the number and type of operands, and tagging the type of functional units required. During the simulation, instructions are executed by the ALUs, MUs, or LTCUs, by calling the evaluation methods of the appropriate instruction definitions.

There is also a set of profiling instructions as listed in Table 3.8, which are the interface between the program being executed and the simulator. The **prb.t** instruction prints out the probe number and the current simulation time. It is useful for measuring the execution length of a program fragment. The **prb.a** and **prb.ai** instructions register an address in a lookup table. During the simulation, when a store (to shared memory) is executed, its target address is checked in the lookup table. If the address has been registered, it will be printed out along with the data to be stored. When a **debug** instruction is encountered by the current head thread, the processor's activities from that point onwards are printed out. The information as to which is reported depends on the level of debugging specified. These instructions do not have delays, but they may slightly affect the performance of some processor components such as FLS, since they are fetched into the instruction streams together with the application instructions and occupy space in the instruction buffers.

Table 3.8 Probing Instructions

Instruction	Description
prb.t *I*	Return probe number *I* and current time.
prb.a *I (\$s1)*	Register address *I (\$s1)* for profiling.
prb.ai *L, I*	Register address *L* + *I* for profiling.
debug *I*	Report processor's activities as specified by switch *I*

3.4.2 Limitations

The simulator runs on a SUN Solaris platform. It reads an input file in the assembly format (ASCII) rather than the binary executable. This allows us to introduce and experiment with new multithreaded instructions without being restricted by the actual instruction set architecture (ISA) and its binary format. However, the simulator does not deal with OS or library calls. When these calls are encountered, they are treated as dummy instructions, i.e. no action is actually executed.

Another limitation is that currently cache hits/misses for the *instruction cache* are not modelled. The cache is always large enough to accommodate the whole input file which is loaded once at the start of the simultion. Similarly, the notion of *data cache* is deliberately omitted. Instead, we refer to the *first-level shared memory* which is also large enough to accommodate the whole program execution.

3.5 The Multithreaded Compiler

Compilers for parallel and multithreaded programs traditionally comprise a front-end source-to-source paralleliser, and a back-end optimiser and code generator. Examples include SUIF [84], Agassiz [79], Polaris [11, 40], and PROMIS [62].

Most front-end parallelising compilers are language-independent and machine-independent. Source programs written in FORTRAN, C, C++, or Java are parsed into universal intermediate representation (IR) format, where structures such as loops and arrays are reserved for parallelisation analyses and transformations. The discovery of parallelism and the extraction of useful information from the sequential code are difficult to be done fully automatically. Interference from programmers is allowed, most commonly via interactive user interface (e.g. SUIF, PROMIS) and the augmentation of source programs with preprocessor directives. The supplementary information can also be provided as external files, as in [29] and [61]. To minimise the degree of machine-dependence, thread manipulating code is typically inserted in the form of function calls which will be later linked to the target-specific multithreaded libraries.

Output from the front-end compiler is fed into the back-end optimiser and code generator. High-level IR is broken down into low-level IR. Propagating high-level information, e.g. alias information and loop-carried data dependence, to the back-end can increase the efficiency of further analyses and optimisations. In SUIF, the information is encapsulated in the annotations attached to SUIF IR objects. In Agassiz, an assertion file is generated to identify the mapping between the front-end and the back-end representations. The back-end compilers are often modified versions of commercial compilers. For instance, Agassiz uses a modified GCC back-end and Polaris uses a modified SGI back-end. The multithreaded libraries are linked to produce the final executable code.

Multiscalar compiler [72] focuses solely on the low-level compilation. A GCC front-end parses, optimises, and compiles a program down to assembly code. The Multiscalar compiler then performs task selection, schedules register communication, and annotates the assembly code with task information. It does not use high-level

structures or information since the task selector processes control-flow graphs at the basic-block level. Furthermore, only register dependence and communication are handled by the compiler, while memory dependence is handled by the hardware.

3.5.1 Compiler Implementation

A multithreaded compiler **threadsuif** was implemented, which analyses sequential programs written in C and automatically transforms them for the multithreaded execution. The SUIF framework [84] was chosen due to its availability and support provided by the distributor (Stanford SUIF Compiler Group). Given SUIF's modular construction and well-defined interface, new functions could be implemented and easily slotted into the compilation flow.

The SUIF compiler system includes a set of compiler *passes* that perform program analyses, optimisations, and transformations on the SUIF intermediate representation (IR). Each pass can be implemented as a separate program. The SUIF IR uses a language-independent abstract syntax tree (AST) representation in two levels:

1. *High-SUIF.* In this level, the AST nodes are high-level control-flow structures such as TREE_IFs, TREE_FORs, or TREE_LOOPs.

2. *Low-SUIF.* In this level, the high-level tree nodes are dismantled into lists of instructions.

Both levels of representation can co-exist. For instance, unstructured control-flow in high-SUIF code is represented by low-level branch and jump instructions. The compiler passes communicate by reading from and writing to SUIF files, where information such as results from the analyses are carried in the annotations attached to the SUIF objects. The SUIF packages [84] used were:

- `basesuif` (version 1.1.2). It is the base system for all other packages.
- `baseparsuif` (version 1.0.0.beta.2). This package includes libraries and passes for parallelisation and dependence analyses, loop transformations, and relevant optimisations.
- `oldsuif` (version 6.0.0). This package is a collection of libraries and passes that work on an earlier generation of the SUIF IR format.
- `suifbuilder` (version 1.0.0). It is an interface for generating SUIF code.
- `suifvbrowser` (version 1.0.0.beta.1). It is a graphical user interface (GUI).
- `tcovsuif` (version 2.0) [85]. This is a contributory package which incorporates profile information into the SUIF code.

The **threadsuif** package comprises of three main modules:

- Multithreaded loop transformer (**loopth**).
- Multithreaded control-speculation transformer (**specth**).
- Multithreaded code generator (**thgen**).

3.5.1.1 Front-end Transformations

The front-end transformers, **loopth** and **specth**, process code in the high-SUIF format. They recognise TREE_FOR and TREE_LOOP structures for the multithreaded loop transformation (Chapter 4), and TREE_IF structures for the control-speculation transformation (Chapter 5). The transformations are described in detail in those chapters.

3.5.1.2 Multithreaded Code Generation

thgen generates the code targeted at the multithreaded architecture. It is a modified version of a MIPS code generator, `mgen`, in the `oldsuif` package. The output files from the front-end transformers are pre-processed by the SUIF passes including `swighnflew`, `oldsuif`, and `mexp`. They reformat the code and prepare information such as register usage for **thgen**. The code generator works in three steps:

1. Translate SUIF instructions into assembly ones and gather the register usage.
2. Allocate saved registers.
3. Determine the size of stack frame and allocate temporary registers.

In our multithreaded model, each thread has a separate register file but they share the same memory space. Therefore, spilling registers to memory is avoided, assuming that there are always sufficient registers in the register file. Moreover, when a procedure call sequence is generated, *infinite-saved-registers* option is used so that the registers are not saved onto stack which is shared by all threads. If the program contains recursive function calls, a private stack should be allocated to each thread. The front-end transformers allocate private memory to threads in the form of arrays and structures which are indexed by the thread numbers, and will be translated into `.data` section in the assembly code.

3.5.2 Compilation Process

An overview of the compilation process is illustrated in Figure 3.7. The analysis in SUIF function is quite simple and may not expose all the parallelism in the programs.

User hints, if needed, can be given in the source files using a pragma directive:

```
#pragma  suif_annote  <annotation name> <information>
                      \_____________hint_____________/
```

When the source code is translated into SUIF IR, the hints will be written as SUIF annotations. Alternatively, annotations can be inserted directly into SUIF files via the graphical user interface, `suifvbrowser`.

The code is pre-processed by various distributed SUIF passes. First, `porky` handles classic optimisations including constant folding, forward propagation, copy propagation, and constant propagation. A new version of C program is generated from the optimised code. It is next compiled with `gcc -a` option, and executed to collect profiling statistics. `tcovsuif` then annotates tree nodes in the SUIF file with the basic block and line counts. The next step analyses high-SUIF structures, such as TREE_FORs, TREE_LOOPs, and TREE_IFs, and determines whether they present good opportunities for the multithreaded and/or speculative execution. Besides detecting parallelisable loops, `skweel` also applies standard transformations such as loop normalisation and loop interchange.

Following the analyses, **loopth** and **specth** recognise the threadable structures and transform them. Cost models can be used to aid the transformation. Ideally, this step should be machine-independent. Practically, however, the transformers are aware of the underlining target architecture and its execution models. After the transformation and code generation (by **thgen**), machine-specific optimisations can be applied, for example, instruction and register communication scheduling.

Finally, the assembly output is supplied to the multithreaded simulator. Profile information is collected. It can be used to help improve the compiling options as well as to adjust the architectural assumptions such as the number of TPUs and ALUs.

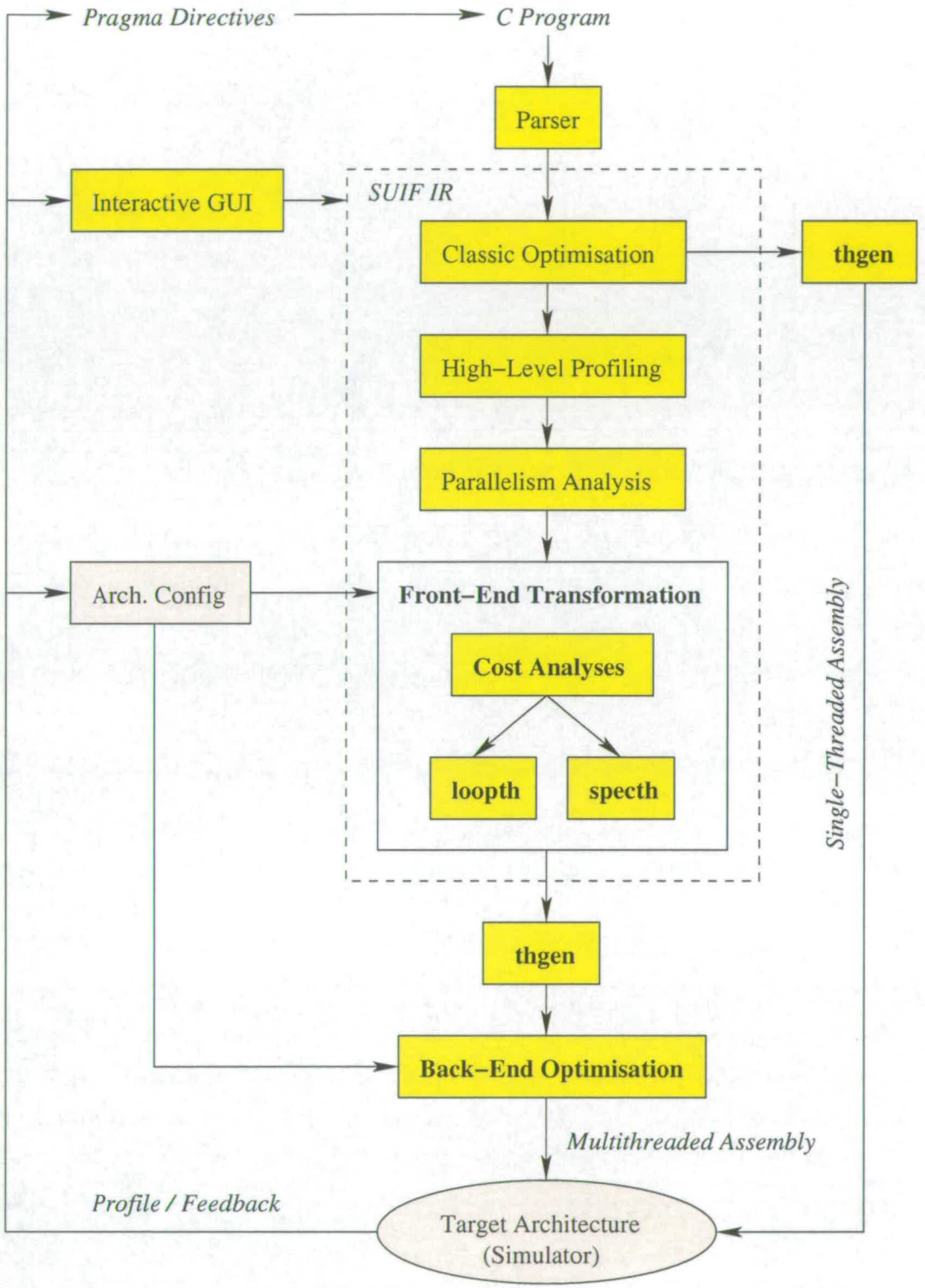


Figure 3.7: An overview of the compilation process

3.6 Discussion

A CMP-based multithreaded architecture has been described, which was enhanced to support hierarchical multithreaded execution, speculation, and register synchronisation and forwarding. The architecture executes the MIPS instruction set which was augmented with multithreaded instructions. There are some similarities and differences between our clustered multithreaded architecture and previous ones, as described next.

During program execution, a thread can dynamically allocate a cluster of slave TPUs to execute a program partition. The number of TPUs in the cluster can be specified in the *cluster allocation* command, which allows TPUs to be used in correspondence to the parallelism in that program partition. This was inspired by the dynamic resource partitioning concept in the Simultaneous Multithreading (SMT) [44]. Each slave thread can also allocate a cluster of slave TPUs at the next level.

Interaction between the master and the slave threads was adapted from architectures such as SPSM, M-Machine, and Superthreaded [18, 23, 68, 78]. Although the slave threads can be related to *subthreads* in [23, 68, 78], the latter typically reside in the master TPU or share hardware resources owned by the master thread. Our slave threads, on the other hand, execute on their own TPUs and transfer results to the master thread after they retire. These results are collected as the cluster's state (or the master's temporary state), and will become the master's current state once the cluster is released. This operation is similar to *merger* in SPSM [18]. An underlining assumption is that the slaves only execute program partitions which are encountered *after* the one executed by the master, according to the sequential semantics. If the program partitions executed by the master and by the slaves are encountered in the reverse order, then the master's current state will be reset to the point as if its execution had not yet started.

The slave threads in the same cluster operate in the predecessor/successor style [45, 65, 68] as follows. Since the slave TPUs are logically connected to each other in a uni-directional ring, a new slave thread is only forked on the next TPU in the ring. After finishing their execution, the slaves must synchronise and retire sequentially.

Hardware support for speculative execution is very simple. Like STAMPede [65], threads can switch between non-speculative and speculative modes - they rely on the software to determine in which mode they should be during the execution. Misspeculation detection and recovery are also performed in the software. Mechanisms are provided for handling a mispredicted thread which include: *interrupting* that thread; *aborting* the slave cluster, if it is a master thread; or switching off the *merger* flag in the *slave retirement* command, if it is a slave thread. Then, the thread can retire and a new one performs the correct execution.

A simple register synchronisation and forwarding mechanism was added to the architecture. The strategy is as follows: a parent thread first executes a command to set *unavailable bits* in the registers, prior to forking a new thread; as the new thread is initialised, *wait bits* in these registers will be automatically set, which enforces synchronisation if it tries to read these registers before they are forwarded from the parent. The set of unavailable registers can be determined by dataflow analysis in the compiler. This idea was borrowed from Multiscalar [12], which declares a set of registers to be written by each task in a *create mask*, and passes this mask to a successor task as an *accum mask*; a task blocks when it tries to read the registers specified in the *accum mask* whose values have not yet been available. The Multiscalar hardware propagates forwarded registers to all the processing units, whereas our multithreaded architecture only forwards registers from the parent to its children.

Because the architectural support for multithreading is kept to the minimum, the

onus is on the compiler to orchestrate the parallelism in programs and specify the execution. New multithreaded instructions were proposed to pass commands from the compiler to the architecture. At run-time, allocating clusters or forking threads are not guaranteed to be successful, depending on the availability of TPUs. In the case of clustering or forking failure, the program will be executed sequentially instead. *Guarded execution* is therefore a key feature in most of the multithreaded instructions. The main idea is to use the result from clustering or forking as *guard* operands in the subsequent multithreaded instructions, to ensure that the program is correctly executed in both sequential and multithreaded modes.

The compiler implemented consists of *source-to-source transformers* for multithreaded loop execution and control speculation, and a *code generator* targetting the multithreaded architecture. The transformers are aware of the underlining architecture and its execution models, which have been summarised and discussed earlier in the section. Then, the code generator generates MIPS instructions combined with the multithreaded ones. The multithreaded loop execution will be described in detail in Chapter 4, and the multithreaded control-speculative execution in Chapter 5.

Chapter 4

Multithreaded Loop Execution

Loops are an important source of parallelism in sequential programs. Loop parallelisation can be performed either statically or dynamically. In a dynamic approach the sequential loops are parallelised at run-time. A static approach, in contrast, transforms the loops at compile-time by inserting thread manipulation routines. Parallelisable loops can be either *do_all* loops, which contain no dependencies between iterations, or *do_across* loops, otherwise. Techniques for testing data dependence can be found in the literature [9, 19, 56, 75, 76]. A number of well-known loop optimisation techniques can be applied prior to the multithreaded transformation in order to expose more loop-level parallelism. For instance, *loop normalisation*, *loop skewing*, and *loop reversal*, rearrange bounds and data dependency pattern in the loops, which enable further optimisations. *Loop interchange* switches the inner and the outer loops in a loop nest - a parallelisable loop can be moved outward to increase the granularity of the loop-level parallelism, or inward to prevent cache overflow. *Loop fission* separates sequential and parallelisable parts of a loop, by breaking a single loop into multiple smaller ones. It is also used to break large loops that do not fit into the cache. *Loop fusion* is the inverse of loop fission, which helps increase instruction-level parallelism in the loops. *Loop*

unrolling can be applied for similar purposes, by replicating the body of the loops. *Strip-mining* and *loop tiling* improve memory locality by dividing the iterations into tiles and traversing between the tiles. *Loop coalescing* and *loop collapsing* transform a loop nest into a single loop, which eliminates the overheads of multiple loops and multi-dimensional array indexing. *Loop peeling* is usually performed in conjuction with the other optimisations as it handles remnant iterations which are leftover from applying other techniques.

The multithreaded transformation is then performed after dependency and parallelism analysis, and this is described next.

4.1 Multithreaded Loop Transformations

An overview of the compilation flow is displayed in Figure 4.1. The transformers detect parallelisable loops in SUIF programs (the SUIF framework was described in Section 3.5); they are TREE_FORs and TREE_LOOPs attached with annotations "do_all" or "do_across". These loops can be detected in SUIF passes such as `skweel`, from pragma directives inserted in the source code, or via the SUIF graphical user interface (GUI). Each parallelisable loop is pre-processed before it is transformed into the multithreaded version. High-level TREE_FORs and TREE_LOOPs constructs are dismantled into straight-line code using functions in the SUIF module, `porky`. Figure 4.2(a) shows a dismantled structure similar to those produced from `porky`, which is next expanded in preparation for the transformation (Figure 4.2(b)). The pre-processing pass also analyses the loop and prepares the following information:

- The number of available TPUs in the processor, which is fixed for all the loops in the program.

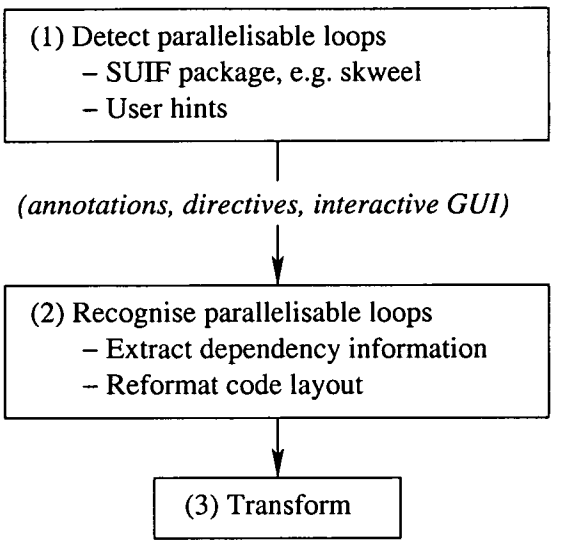


Figure 4.1: An outline of the loop transformation

- The number of slave threads to execute the loop in parallel, which is obtained from the cost analysis.
- The lists of instruction pairs that may cause loop-carried dependence. Sources and sinks of the dependency edges are maintained in `strlist[num_dep_pairs]` and `lodlist[num_dep_pairs]` respectively.
- Exit points. The natural exit of a loop is after the continuation test or `BRK_LABEL` in Figure 4.2(a). Other exit points may also be present inside the loop body.

Two loop transformation algorithms are described in Sections 4.1.1 and 4.1.2, respectively. **Loop_Transformer_1** transforms loops with only natural exits while **Loop_Transformer_2** transforms loops with multiple exits. The appropriate algorithm is chosen automatically for each loop by the pre-processing routine.

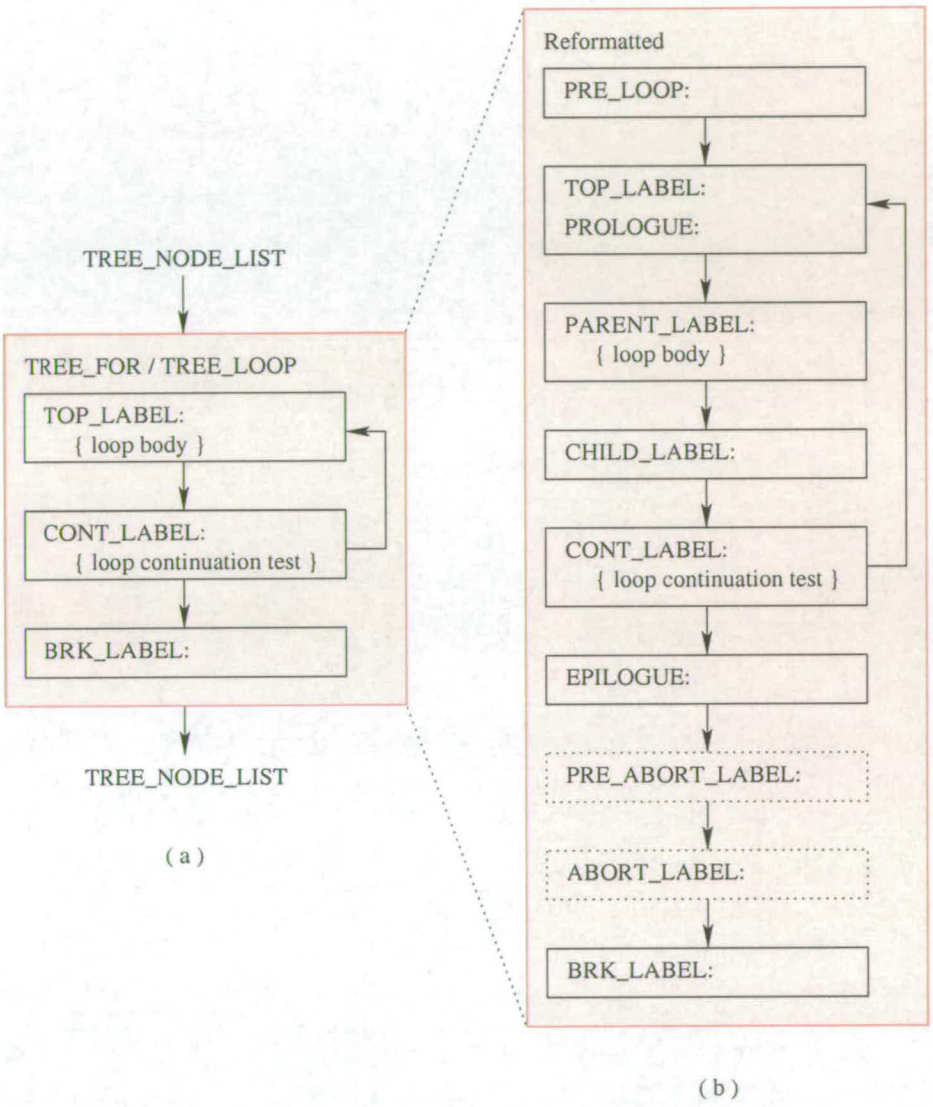


Figure 4.2: Loop structure in SUIF IR, (a) before and (b) after loop expansion

```
 1   int guard[NUM_TPUS];                          // working variables
 2   int readsynch, csucc, xsucc, ysucc, merge;    // working variables
 3   int myself, mychild, myparent;                // working variables
 4   for (i = 0; i < NUM_TPUS; i++) guard[i] = 1;  // initialise guards
 5   PRE_LOOP:                                     //
 6       readsynch = csucc = ysucc = 0;            //
 7       csucc = cform (NUM_SLAVES);               // form a slave cluster
 8       myself = adr ();                          // get self's address
 9       guard[myself] = csucc;                    //
10       ysucc = yfrk (guard[myself], TOP_LABEL);  // fork the first slave
11       mychild = cadr ();                        // get child's address
12       if ( ysucc ) {                            //
*13          sstp (0, mychild);                    // send synchronisation signal to 1st slave
14           ... independent works ...             // from code motion
15           crels (guard[mychild], 1);            // release the cluster
16       }                                         //
17       else                                      //
18           ... independent works ...             // the same copy as line 14
19   TOP_LABEL:                                    // start loop execution
20   PROLOGUE:                                     //
21       xsucc = 0;                                //
22       myself = adr ();                          // get self's address
23       xsucc = xfrk (guard[myself], CHILD_LABEL); // fork the next slave
24       mychild = cadr ();                        // get child's address
25   PARENT_LABEL:                                 //
26       ... original loop body ...                //
27       merge = 1;                                // useful execution completed
28       if ( xsucc ) goto EPILOGUE;               //
29       else {                                    //
30           readsynch = 0;                        //
31           goto CONT_LABEL;                      //
32       }                                         //
33   CHILD_LABEL:                                  //
34       readsynch = 1;                            //
35       merge = 0;                                //
36   CONT_LABEL:                                   //
37       ... original loop test ...                // branch to TOP_LABEL if continue
38   EPILOGUE:                                     //
39       myself = adr ();                          // get self's address
40       xstp (guard[myself], merge);              // retire slave thread
41   BRK_LABEL:                                    // natural exit
```

Figure 4.3: Multithreaded loop generated by **Loop_Transformer_1**

4.1.1 Simple Loops

Thread manipulation code is inserted in each block of the reformatted loop (Figure 4.2(b)). Figure 4.3 shows the code outline of a multithreaded loop with the only exit at BRK_LABEL. Private variables can be allocated via arrays, e.g. guard[NUM_TPUS], and indexed by the thread addresses. They may also live in registers, provided that the number of registers is sufficient to prevent spilling to the shared memory.

Master Thread

The execution starts at PRE_LOOP. The master thread attempts to form a slave cluster (line 7) and stores the result in guard. If the operation succeeds, then the loop will be executed by the slaves; otherwise, it will execute the loop itself. Due to the guard operands of subsequent multithreaded instructions (e.g lines 23 and 40), they are treated as nop instructions when executed by the master.

Independent computation before or after the loop can be inserted ahead of crels (line 15) by code motion technique, and executed in parallel with the main loop execution. When the master reaches crels, it sends a synchronisation signal to the slaves and waits until they retire. Then it transfers temporary register updates by the slaves to its register file and frees the cluster. Since the program counter of the last slave becomes the current program counter of the master, the execution will resume at BRK_LABEL (line 41) which is the exit point of the original loop.

Slave Threads

The execution starts at TOP_LABEL (line 19). A new thread is forked (line 23) before the current thread continues to execute the loop body. At the end of the execution, the merge flag is set (line 27) indicating that the register updates by this thread will

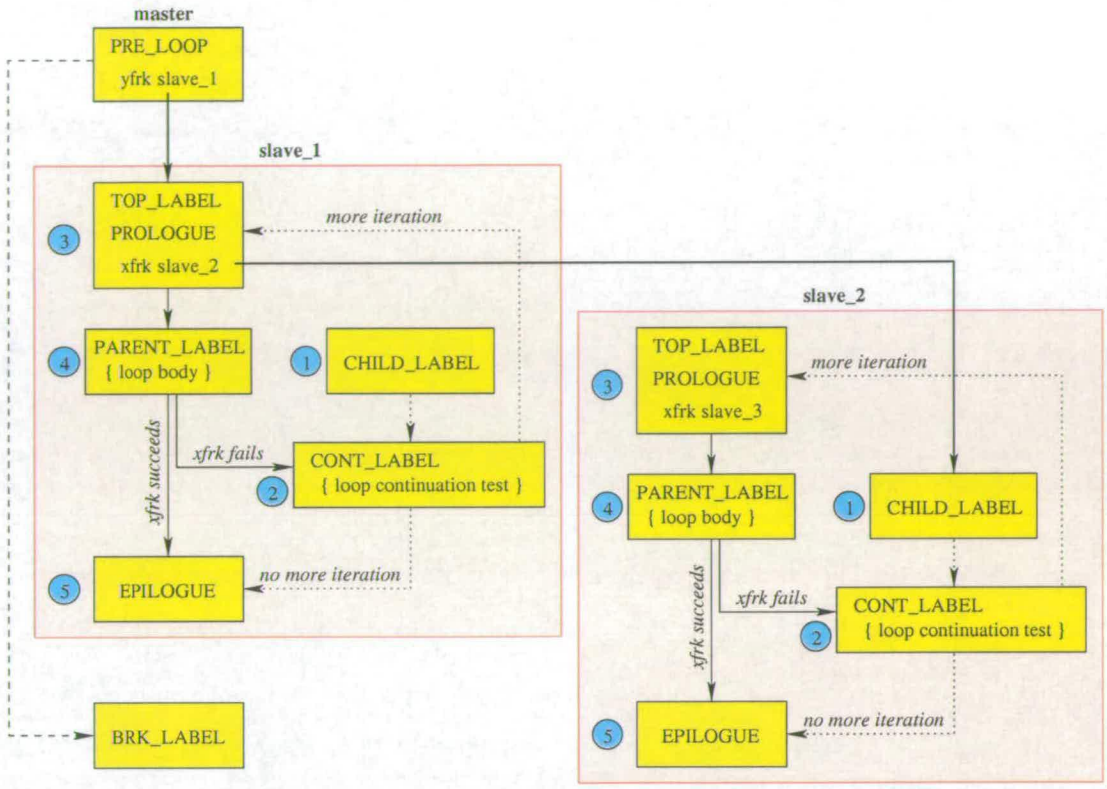


Figure 4.4: Diagram of the multithreaded loop in operation

be merged into the master's temporary state. If the fork is successful, then the current thread waits to retire and passes the synchronisation signal to its child (line 40). Otherwise, it will perform the loop continuation test and execute the next iteration itself.

The child's execution begins at `CHILD_LABEL` (line 33). The loop continuation test (line 36) is performed early to determine whether to start a new iteration, thereby limiting the amount of speculative work of the child thread. If the test fails, the child only waits to synchronise with its parent and retires without merger. Figure 4.4 depicts the multithreaded loop in operation. Life cycles of the slave threads, except the first one which is spawned by the master (the first slave thread starts at `TOP_LABEL`), start from

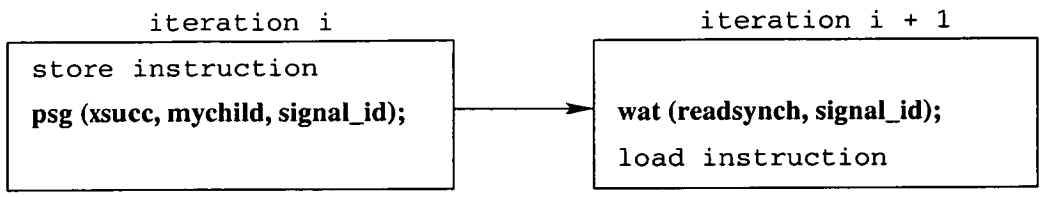


Figure 4.5: Store/load synchronisation in **Loop_Transformer_1**

childhood which are paths represented by dotted lines. When a child thread reaches PROLOGUE and executes the `xstp` instruction, it becomes a parent and follows the paths represented by the solid lines. From the diagram, the master thread follows the paths represented by the dashed line.

Since every thread encountering an `xstp` is blocked until the signal is received from its parent, the master thread executes `sstp` (line 13) in order to pass the signal to the first slave prior to its own computation. This permits the slave TPUs to be reused by multiple slave threads, which is called *recycling* execution. However, this is impractical for nested loops. As a thread executing an outer loop iteration tries to pass the signal to its slaves, it may be blocked waiting for the signal from its own parent (since the signal is forced to pass around in the correct order, the thread will eventually be unblocked without causing any deadlock). In such cases, the `sstp` instruction is excluded since the execution is *non-recycling*. The non-recycling execution will be discussed later in the chapter.

Data Dependence

For each pair of dependent instructions in `strlist` and `lodlist`, synchronisation is enforced by passing and waiting for a signal, as shown in Figure 4.5. The `signal_id` is an integer value unique to each store/load pair; `psg` is a non-blocking send while `wat` is

a blocking receive. The execution of `wat` is guarded by `readsynch`, whose value is set when a new thread is created (line 34 in Figure 4.3). If a thread has to execute the next iteration itself, i.e. either `xfrk` fails or it is the master thread, then `readsynch` must be switched off (line 30 in Figure 4.3), since the new iteration need not synchronise its memory operations with the previous iteration executed by the same thread. On the other hand, `psg` is guarded by `xsucc` whose value is set if the thread successfully forks a new thread.

4.1.2 Loops with Multiple Exits

Loop_Transformer_2 (Figure 4.6) operates on loops with multiple exit points. At present, it automatically handles `break` statements in C which are embedded in the loop body. Thread manipulation code for `PRE_LOOP`, `PARENT_LABEL`, and `CHILD_LABEL` is the same as before with modification being made to `PROLOGUE` and `EPILOGUE`, and the additional `PRE_ABORT_LABEL` and `ABORT_LABEL` were introduced to handle speculation.

Speculation Handling

Although the loop continuation is tested early when a new thread is spawned, the iteration may still be invalid if a thread executing any previous iteration encounters an exit point. Therefore, a newly-created thread has to turn off the *safe* flag and become speculative (line 23). Subsequent stores by the thread will be buffered in its private memory and only committed to the master thread's temporary buffer before it retires (lines 39 and 40).

The original `breaks` in the source program were translated to jumps to `BRK_LABEL` in the SUIF code. The targets of these jumps were changed to `PRE_ABORT_LABEL` by

the transformer. When the first thread encountering an exit branches to this label, it interrupts the child's execution (line 45) and commits its speculative stores and register updates up to the break point (lines 46 and 47) before retiring. The subsequent threads are recursively interrupted (line 52) and retire without merger (line 53). If the interrupted thread governs the multithreaded execution of the inner loop, it also aborts its slave cluster (lines 44 and 51).

Suppose that the following loop is transformed for the multithreaded execution:

```
int a[5] = {1, 2, 3, 4, 5};
int b[5] = {6, 7, 8, 9, 10};
for (int i = 0; i < 5; i++) {
    if (i < 3)
        a[i] = ( (a[i] * 111) + (b[i] * 222) ) * 333;
    if (i > 0) break;
}
```

The compiler will generate 5 slave threads $\{T_0, T_1, T_2, T_3, T_4\}$ to execute the loop iterations (with induction variable `i` = {0, 1, 2, 3, 4}, respectively). As T_3 bypasses the computation under the first condition and arrives at `break` before the others, it will interrupt T_4 and try to commit its result. The `xstp` and `cmmt` instructions force the threads to wait until they are signalled by their predecessors. The next thread that encounters `break` is T_1. It interrupts T_2, which, in turn, interrupts T_3 from blocking at the `cmmt` instruction. Eventually, the threads will commit and retire in the correct sequential order despite exiting the loop out-of-order, as shown in Table 4.1.

```
 1   int guard[NUM_TPUS];                          // working variables
 2   int readsynch, csucc, xsucc, ysucc, merge;    // working variables
 3   int myself, mychild, myparent;                // working variables
 4   for (i = 0; i < NUM_TPUS; i++) guard[i] = 1;  // initialise guards
 5   PRE_LOOP:                                     //
 6       same as Figure 4.3, lines 6-18            //
19   TOP_LABEL:                                    // start loop execution
20   PROLOGUE:                                     //
21       xsucc = 0;                                //
22       myself = adr ();                          // get self's address
*23      safe (guard[myself], 0);                  // become speculative
24       xsucc = xfrk (guard[myself], CHILD_LABEL); // fork the next slave
25       mychild = cadr ();                        // get child's address
26   PARENT_LABEL:                                 //
27       same as Figure 4.3, lines 26-32           //
33   CHILD_LABEL:                                  //
34       same as Figure 4.3, lines 34-35           //
36   CONT_LABEL:                                   //
37       ... original loop test ...                // branch to TOP_LABEL if continue
38   EPILOGUE:                                     //
*39      if ( guard[myself] ) cmmt (merge);        // commit/discard speculative stores
40       xstp (guard[myself], merge);              // retire slave thread
41       goto BRK_LABEL                            //
*42  PRE_ABORT_LABEL:                             //
43       if ( guard[myself] ) {                    //
44           crels (in_csucc, 0);                  // abort inner-level cluster
45           isg (xsucc, mychild, ABORT_LABEL);    // interrupt child's execution
46           cmmt (1);                             // commit/discard (current thread)
47           xstp (guard[myself], 1);              // retire slave thread
48       }                                         //
49       goto BRK_LABEL;                           //
*50  ABORT_LABEL:                                 //
51       crels (in_csucc, 0);                      // abort inner-level cluster
52       isg (xsucc, mychild, ABORT_LABEL);        // interrupt child's execution
53       xstp (guard[myself], 0);                  // retire without merger
54   BRK_LABEL:                                    // natural exit
```

Figure 4.6: Multithreaded loop generated by **Loop_Transformer_2**

Table 4.1 Order of commit and retirement

Thread	Action	Code Executed (line in Figure 4.6)
T_0	commits	line 39
T_0	retires	line 40
T_1	commits	line 46
T_1	retires	line 47
T_2	retires	line 53
T_3	retires	line 53
T_4	retires	line 53

Master Thread Execution

As discussed in Section 4.1.1, the master thread suspends at the `crels` instruction. Some slaves commit their speculative stores to the master's temporary buffer. The data from this buffer is transfered to the working speculative buffer before the master thread resumes its execution at `BRK_LABEL`. If the loop is in a nest, then the master commits these results in the `EPILOGUE` of the outer loop iteration. For the outermost loop, the results collected from all the threads in the system are committed after the master exits at `BRK_LABEL`.

Data Dependence

Data dependence between iterations is handled in much the same way as before. However, being speculative, each slave reads from the speculative buffer of its predecessor instead of from the shared memory. The `psg/wat` instruction pair (Figure 4.5) ensures that the consumer waits until the most recently-updated data is available. Figure 4.7 shows an example of multithreading in nested loops, which assumes that there is data

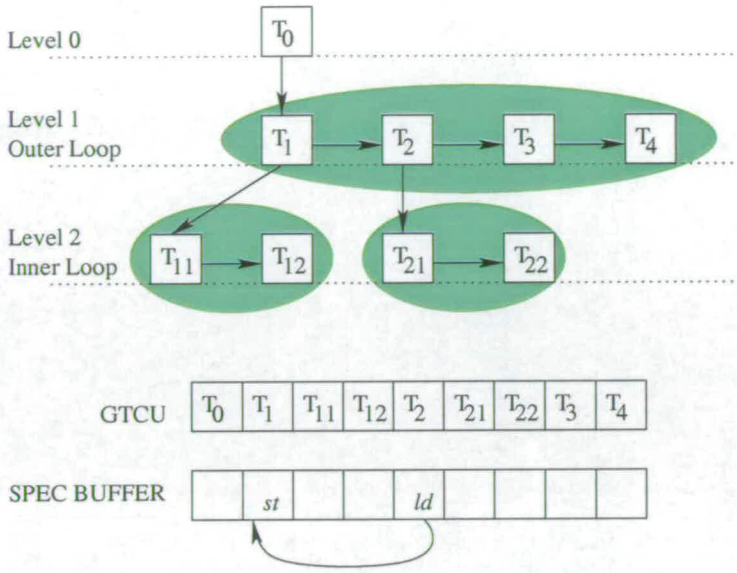


Figure 4.7: Nested loop execution in speculative mode

dependence between the outer loop iterations. Following the thread order maintained by the global thread control unit (GTCU), T_2 retrieves the data from T_1's buffer after the synchronisation. Our transformation is applied to each loop separately. Therefore, data dependence between iterations of different loops, such as between T_{12} and T_2, is not recognised by the transformer. In those cases, only one loop is chosen for the multithreaded execution.

Arbitrary Exits

In the case of arbitrary exits other than `break` statements, the first thread reaching an exit saves the target address before jumping to `PRE_ABORT_LABEL`. Instructions are inserted at the end of the aborting sequence, i.e. after line 47, to load and branch to the target address after the execution is resumed by the master thread.

This transformation is also applicable to loops from the previous section. Because no exit is found in the body of such loops, PRE_ABORT_LABEL and ABORT_LABEL blocks are never executed.

4.1.3 Register Communication

Data dependence between threads can also be handled by register forwarding. This approach requires dataflow analysis in the assembly level or after the register allocation phase. As a result, when this option is selected, the front-end transformers only mark places where data dependencies might occur. The actual instructions, `uregs` and `fregs`, will be inserted during the code generation phase. The multithreaded transformation of the following loop is next considered:

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total = 0;
for (int i = 0; i < 10; i++)
    total = total + a[i] * 2;
```

Loop-carried data dependence exists due to the reads and writes of the variable `total`. Figure 4.8 shows the assembly code of the multithreaded loop when memory communication is used. The result of the summation (line 20) is stored in the shared memory and loaded by the next iteration. Synchronisations are required before the load (line 18) and after the store (line 22).

```
 1          ...                          //
 2   L2.main:                            // TOP_LABEL
 3   L3.main:                            // PROLOGUE
 4          li     $75, 0                //
 5          adr    $76                   //
 6          muli   $11, $76, 4           //
 7          lai    $12, __S1.main, 0     //
 8          addu   $13, $12, $11         //
 9          lw     $73, 0($13)           // load guard value
10          xfrk   $73, $75, L5.main     //
11          cadr   $77                   //
12   L4.main:                            // PARENT_LABEL (loop body)
13          muli   $14, $69, 4           //
14          la     $15, 64($29)          //
15          addu   $24, $15, $14         // calculate address of a[i]
16          lw     $25, 0($24)           // load a[i]
17          muli   $80, $25, 2           // $80 ← a[i] * 2
*18         wat    $79, 1                // synchronise load
19          lw     $8, 104($29)          // load total
20          add    $9, $8, $80           // $9 ← total + $80
21          sw     $9, 104($29)          // store result
*22         psg    $75, $77, 1           // synchronise store
23          beqz   $75, L8.main          //
24   L10.main:                           // xfrk succeeds
25          j      L6.main               //
26   L8.main:                            // xfrk fails
27          li     $79, 0                // turn off wat's guard
28          j      L0.main               //
29   L5.main:                            // CHILD_LABEL
30          li     $79, 1                // turn on wat's guard
31   L0.main:                            // CONT_LABEL
32          addi   $69, $69, 1           //
33          li     $10, 10               //
34          bgt    $10, $69, L2.main     // continuation test
35   L6.main:                            // EPILOGUE
36          ...                          //
```

Figure 4.8: Transformed loop using memory communication

```
 1          ...                          //
 2   L3.main:                            // PROLOGUE
 3          ...                          //
 4          li      $45, 2               // offset
*5          uregs   $73, 64($45)         // set $70 unavailable
 6          xfrk    $73, $75, L5.main    //
 7          ...                          //
 8   L4.main:                            // PARENT_LABEL (loop body)
 9          muli    $14, $69, 4          //
10          la      $15, 64($29)         //
11          addu    $24, $15, $14        // calculate address of a[i]
12          lw      $25, 0($24)          // load a[i]
13          muli    $80, $25, 2          // $80 ← a[i] * 2
14          # [``synch_read'' 1]         // annotation added by transformer
15          # [``synch_write'' 1]        // annotation added by transformer
16          add     $70, $70, $80        // $70 ← $70 + $80
*17         fregs   $75, 64($45)         // set $70 available and forward
18          beqz    $75, L8.main         //
19          ...                          //
```

Figure 4.9: Transformed loop using register communication

Figure 4.9 shows assembly code of the same loop when register communication is used. In our example, the source and sink of the dependency edge point to the same instruction (line 16). Lines 4, 5, and 17 are added after the code generation as it is when the register dependence (caused by $70) is known. The register identifier of $70 is encoded as

$$70 = 6 + 32(2) \quad , \quad \begin{array}{lcl} \text{base register} & = & 6 \\ \text{mask} & = & \text{0x00000040 or 64} \\ \text{offset} & = & 2 \end{array}$$

The `uregs` instruction is inserted before `xfrk` to enforce synchronisation if the child thread tries to read $70 before it is available. The instruction is guarded by the same condition as `xfrk`. After the register value is produced, it is forwarded to the next thread by `fregs` which is guarded by the result of the fork. Figure 4.10 depicts the register communication between threads.

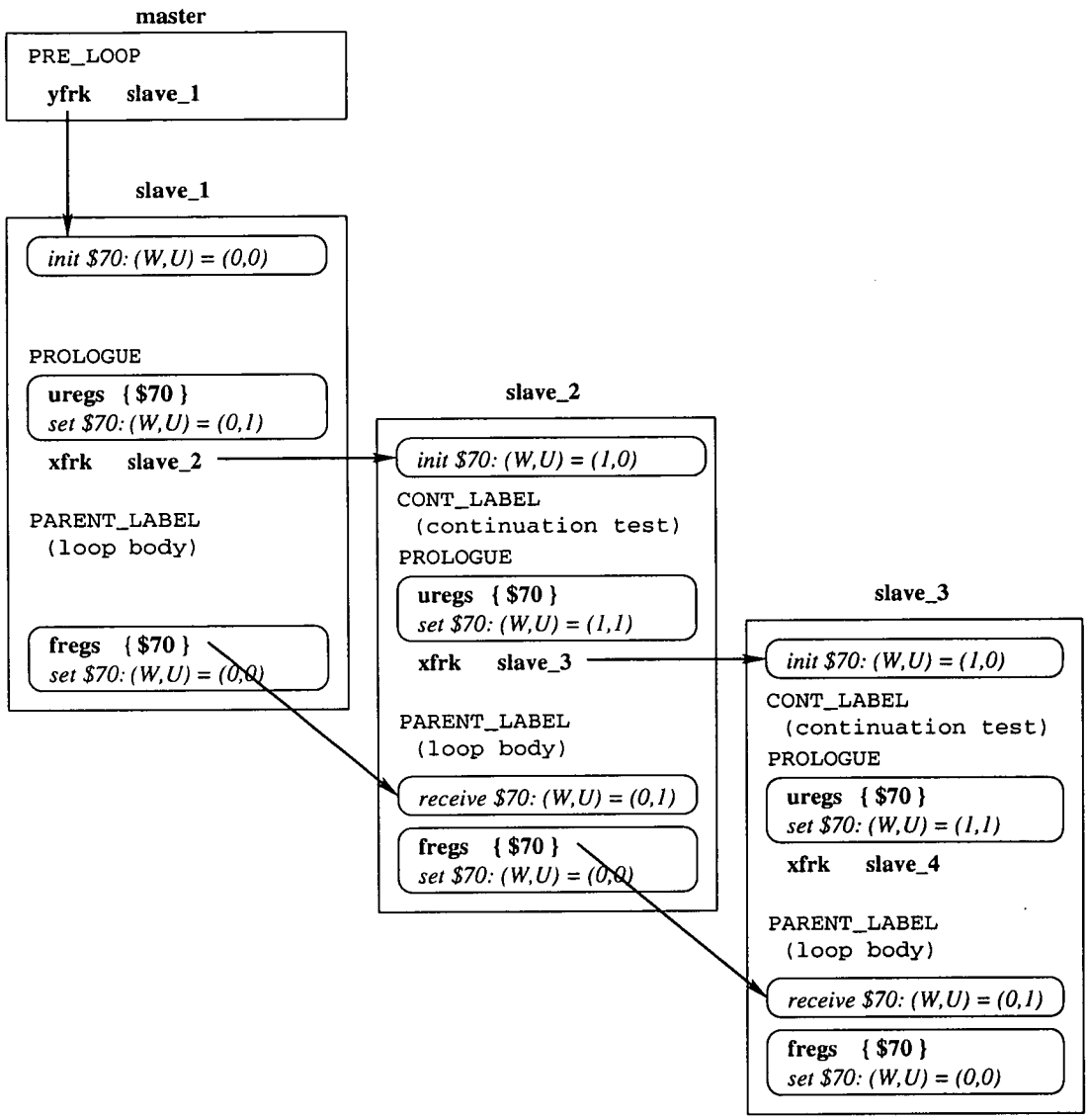


Figure 4.10: Diagram of register communication for register $70

4.2 Performance Evaluation

This section reports results of executing sequential and multithreaded programs on our simulated architecture. The architectural details and compilation framework were described in Chapter 3. Optimisations performed by SUIF prior to the multithreaded transformation are classic optimisations (constant folding, forward propagation, copy propagation, and constant propagation) and basic loop optimisations (loop normalisation, loop skewing, and loop reversal). Sequential programs were transformed using **Loop_Transformer_1** and **Loop_Transformer_2** described earlier. Techniques such as loop unrolling and loop peeling were also explored.

4.2.1 Benchmarks

The C version of the Livermore kernels [82] were used as benchmarks in the experiments. The kernels are placed in separate programs which consisted of three phases: initialisation, main computation of the kernel, and verification. Each kernel is executed repetitively enough to dominate the total execution time of the program.

The statistics for the Livermore kernels are summarised in Table 4.2. The parameters K and I were taken from the full benchmark version [81] (the first set of DO-loop spans), except *matrix multiplication (U_21)* which was scaled down to correspond to the same workload as the other kernels. The last four columns present statistics of the benchmarks from the sequential execution.

Table 4.2 Benchmark description and general statistics

Name	Kernel Description	Kernel Statistics			Dynamic	Distribution (%)		
		K	I	K * I	Instructions	Init	Main	Verify
A_1	Hydrodynamic code	70	1,001	70,070	1,779,326	0.64	99.03	0.32
C_3	Inner product	90	1,001	90,090	1,183,761	0.95	99.05	0.00
D_4	Banded linear equations	140	600	84,000	1,546,251	0.83	99.17	0.00
F_6	General linear recurrence equation	30	1,954	58,620	1,240,448	2.85	97.12	0.03
G_7	State equation	40	995	39,800	2,616,272	3.08	96.70	0.22
H_8	Alternating direction, implicit integration code	100	198	19,800	3,489,128	1.09	98.67	0.24
I_9	An integration predictor	360	101	36,360	2,019,297	0.97	98.99	0.04
J_10	A difference predictor	340	101	34,340	2,649,431	1.18	98.35	0.46
L_12	First difference	120	1,000	120,000	2,060,662	0.75	98.92	0.33
N_14	1-D particle-in-cell code	20	2,000	40,000	1,450,765	0.82	99.15	0.03
R_18	2-D explicit hydrodynamic code	20	495	9,900	3,035,697	1.33	98.32	0.35
U_21	Matrix multiplication	5	15,625	78,125	1,849,569	0.84	98.89	0.26
V_22	Planckian distribution procedure	70	1,001	70,700	2,415,445	0.79	98.96	0.25
				Average	2,102,773	1.24	98.56	0.20

K : the number of times a kernel is executed.

I : the number of iterations executed in a kernel execution.

$K * I$: the total number of iterations executed.

Table 4.3 Parameters for the simulated multithreaded architecture

Configuration Sizes		Latencies (in time units)	
instruction buffer (inst.)	10	ALU multiply	12
total TPUs	18	ALU divide	20
ALUs/TPU	2	ALU others	4
registers/TPU	120	MU load/store	4
		LTCU queries (group 2)	2
		LTCU others	4
		buffer hit	1
		buffer miss	5

Table 4.4 Multithreading overheads

Overheads	Routine	Average Time units
Master's	PRE_LOOP	50
Slave's : Loop_Transformer_1		
Fork / Initialisation	PROLOGUE and CHILD	50
Retirement	EPILOGUE	42
Slave's : Loop_Transformer_2		
Fork / Initialisation	PROLOGUE and CHILD	54
Retirement	EPILOGUE	50
	PRE_ABORT	20
	ABORT	12

Table 4.5 Details of parallelisable loops in the benchmarks

Program	A_1	C_3	F_6	G_7	H_8	I_9	J_10	L_12	N_14	V_22
Loop-carried Dependence	X	√	√	X	X	X	X	X	X	X
Body Length (time units) [a]	132	73	68	358	749	266	317	75	173	169

[a] From sequential execution, ALUs = 2.

4.2.2 Results and Discussions

The first experiment compared the performance of multithreaded programs to their sequential version. Parameters used in the simulation are listed in Table 4.3. The sequential programs had been optimised using classic optimisations and executed on the architecture with the number of ALUs ranging from 1 to 4. It was observed that most programs used at most 2 ALUs. Although some programs used 3 or 4 ALUs, the utilisation of those extra ALUs were quite low. Thus, the number of ALUs per TPU in the table is set to 2.

All the benchmarks except *D_4*, *R_18*, and *U_21* contain one-level nested parallelisable loops. They were transformed into multithreaded code, for cluster sizes ranging from 2 to 16, in steps of 2. Because the benchmarks have only natural exits, we tested **Loop_Transformer_2** by re-writing the loops using `while(TRUE)` instead of the original `for( ... )`, and `break` once the loop index value exceeds the upper bound . The execution times of both versions were similar, and therefore the results from the original loops were reported in Figure 4.11. The multithreading overheads in Table 4.4 are the average execution time of the thread manipulation routines, which were measured from the experiments in this section.

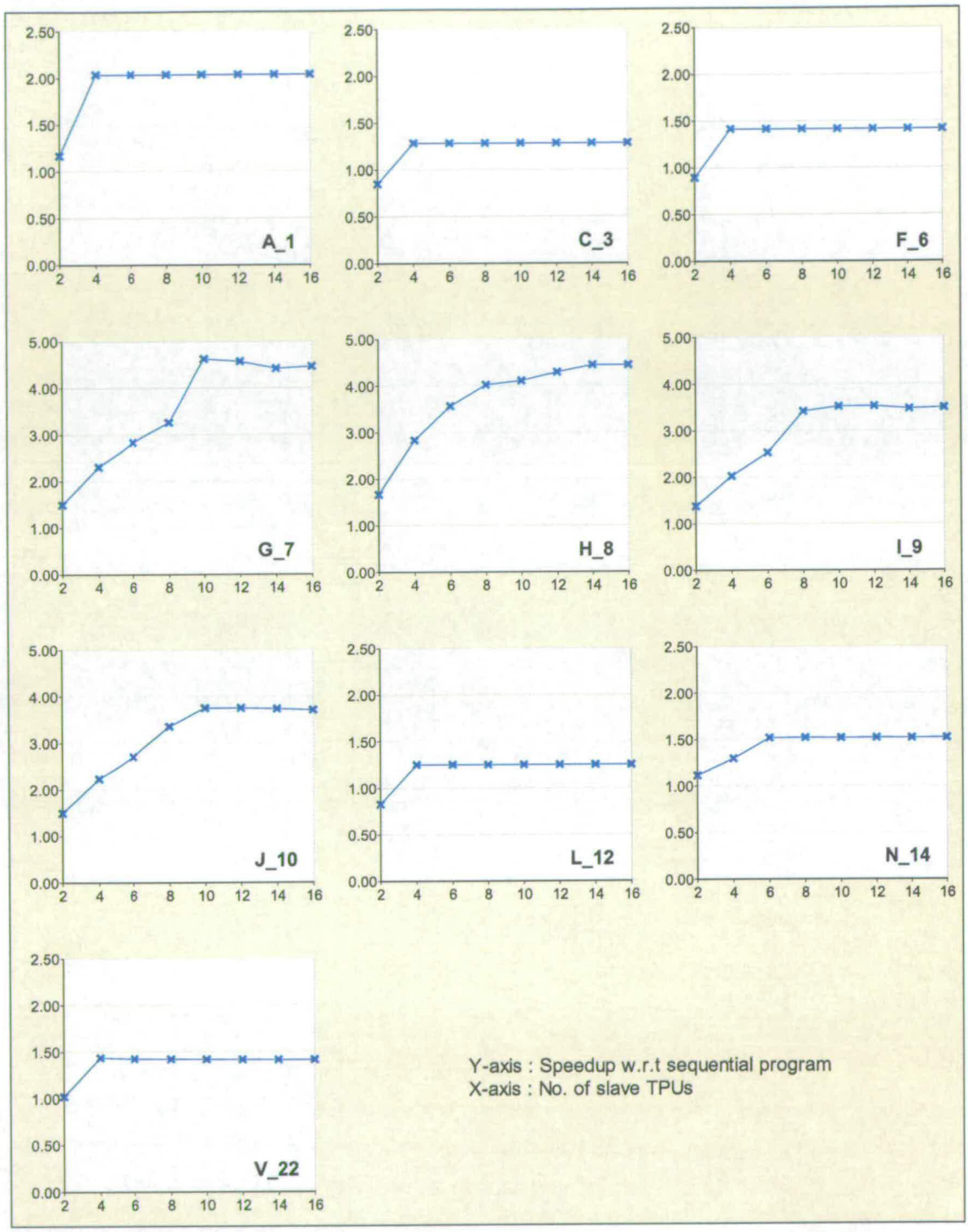
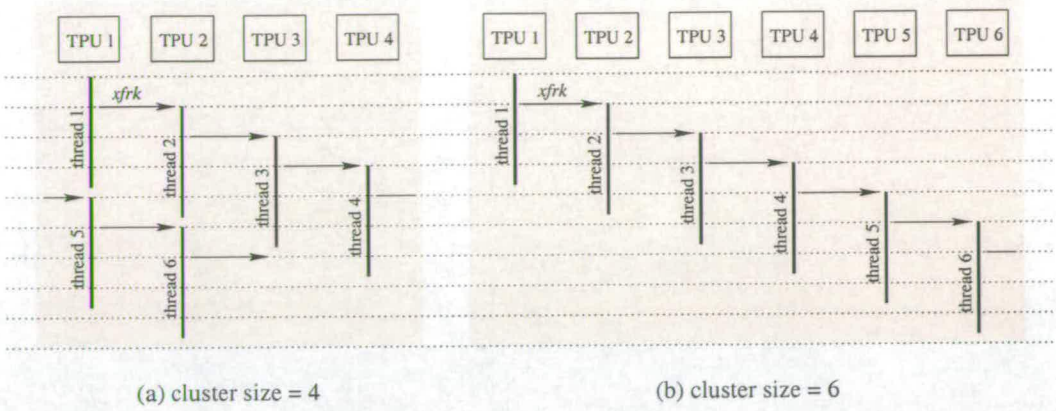


Figure 4.11: Speedup of multithreaded programs with cluster size ranging from 2 to 16 TPUs, in steps of 2



(a) cluster size = 4 (b) cluster size = 6

Figure 4.12: A saturation point being reached at cluster size = 4

Loop bodies of the loops in *C_3*, *F_6*, and *L_12* are fairly short. The ones in *C_3* and *F_6* also contain loop-carried data dependence. The multithreaded versions of these benchmarks deliver little speedup over the sequential programs; they are even worse when the cluster size is 2. In contrast, the multithreaded execution offers good speedup in *G_7*, *H_8*, *I_9*, and *J_10* and the loop bodies in these benchmarks are reasonably large (see Table 4.5). The speedup generally levels off after 8-10 slave TPUs. The slave TPUs are recycled among slave threads. Due to the inherent parallelism and the execution pattern of the loops, after certain points, their performance will no longer improve in spite of the increase in the number of TPUs. Figure 4.12 depicts an example. As mentioned in Section 3.4.1.3, the simulated architecture assumes that the bus delay is included in the delays of the other processor components such as ALUs, and bus contention is lumped in the contention for these resources. Communication delay was assumed to be uniform since the communication is only permitted between parent and child threads which are likely to execute on neighbouring TPUs.

Table 4.6 Details of parallelisable nested loops

Program	from outermost to innermost loops	
	Iterations	Body Length (time units) [a]
D_4	(3, 194)	(17633, 67)
R_18	(5, 100) (5, 100) (5, 100) [b]	(51116, 497) (75174, 720) (12586, 108)
U_21	(25, 25, 25)	(72489, 2881, 95)

[a] From sequential execution, ALUs = 2.

[b] 3 sets of nested loops executed sequentially.

In the next experiment, multithreaded execution in nested loops in *D_4*, *R_18*, and *U_21* were performed (the details of these loops are shown in Table 4.6). The speedup of the multithreaded versions of *D_4*, *R_18*, and *U_21* across different numbers of TPUs are shown in Figures 4.13 and 4.14. The nested loop execution is labelled as follows: $N(2,4)$, indicats that 2 and 4 slave TPUs are allocated to the outer and the inner loops, respectively. *OUTER*, *MID*, or *INNER* represent the multithreaded execution in the outermost, middle, or innermost loops only. The total number of TPUs in the graphs includes the master and the slave TPUs.

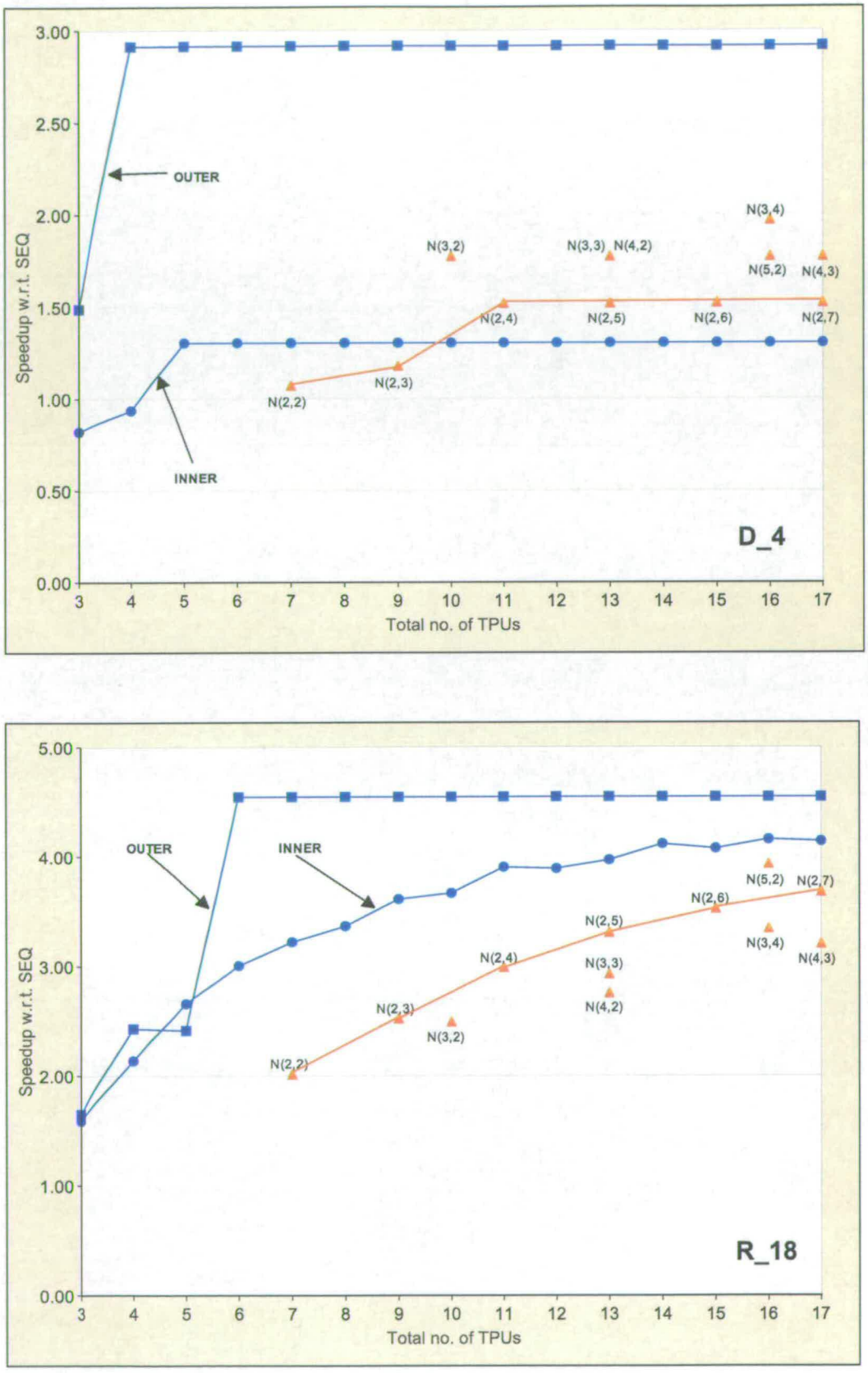


Figure 4.13: Speedup of multithreaded versions of *D_4* and *R_18*

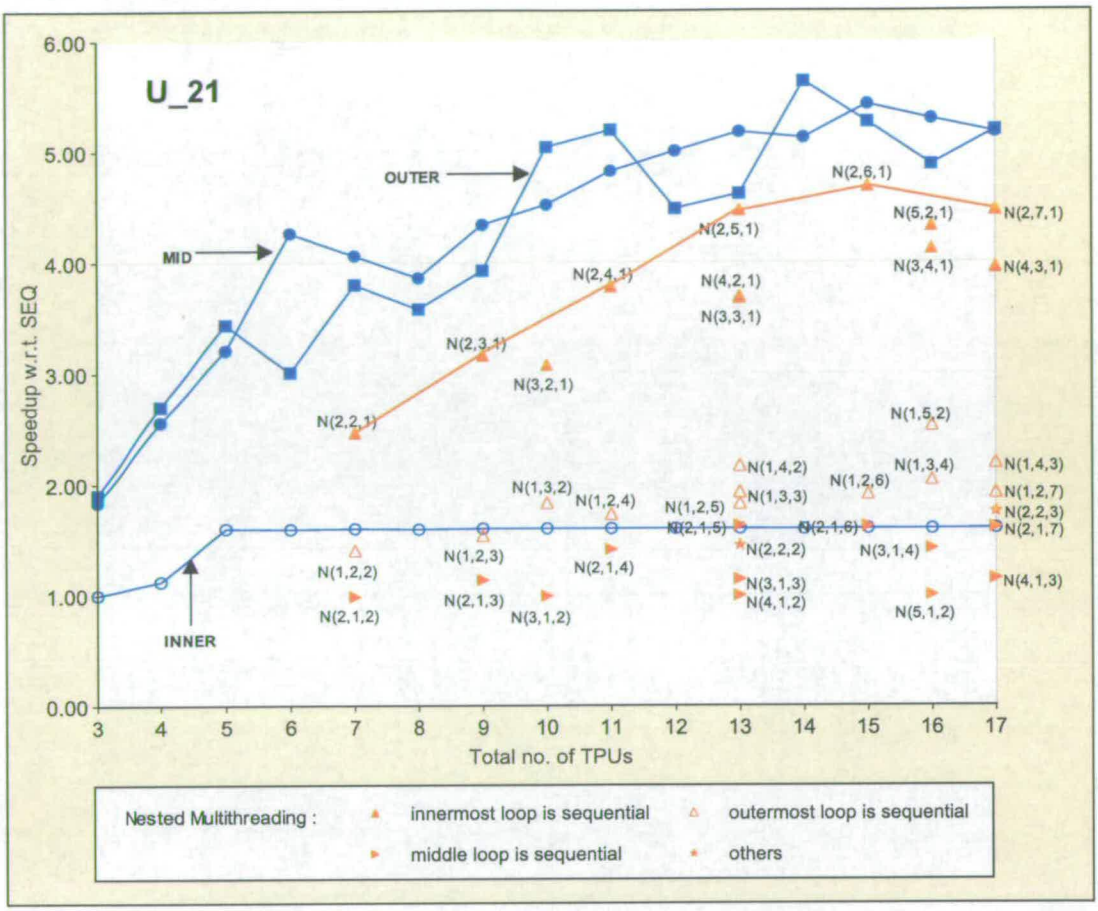


Figure 4.14: Speedup of multithreaded versions of *U_21*

For all the benchmarks, two-level multithreading yields no better performance than one-level multithreading in the outermost or middle loops. In the case of one-level multithreading in the innermost loops, it appears that the loop bodies in these benchmarks are too small for the multithreaded execution to be beneficial. A drawback of the multithreading method as mentioned earlier is that at the start of each iteration, a fork instruction is executed and is only successful if the next slave TPU is available. A thread occupies a TPU, even though its execution has completed, until it receives the synchronisation signal from its predecessor, allowing it to commit and free the

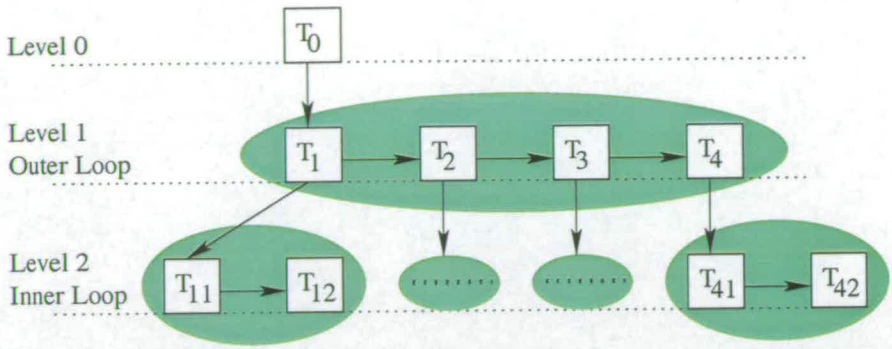


Figure 4.15: An example of nested multithreading

TPU. An example is shown in Figure 4.15. In cluster $\{T_{41}, T_{42}\}$, the first slave thread (T_{41}) waits for the signal from its master (T_4) which, in turn, awaits the signal from its preceding master (T_3). As a result, after a few iterations are executed in parallel, the remaining ones are executed sequentially because no thread is further sparked.

One approach to this problem is to enhance the architecture, by differentiating the synchronisation between global and local levels so that the clusters can be managed fully independently from each other. The solution proposed in this thesis is to use compile-time techniques such as loop unrolling and loop peeling to improve performance of the multithreaded programs. This approach was chosen as it does not require any alteration to the architectural design. In addition, since the current architecture permits threads to commit and retire one-by-one, it simplifies the study of control-speculative execution which will be described in Chapter 5.

4.2.2.1 Loop Peeling

Loop peeling removes a small number of iterations from either the beginning or the end of a loop and executes them separately. A common use is to remove data dependencies

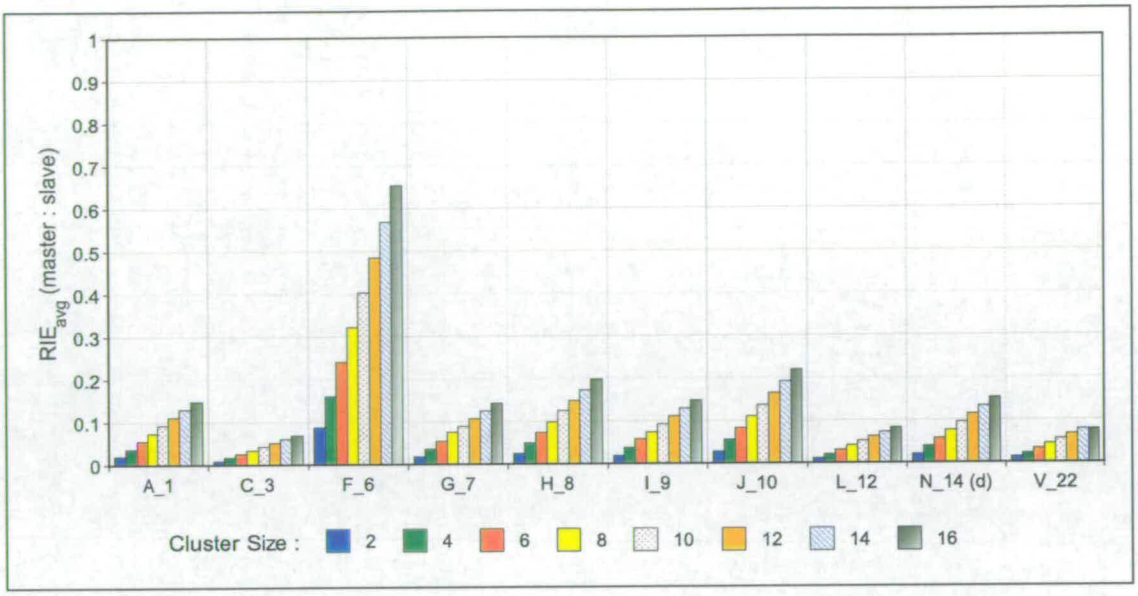


Figure 4.16: RIE_{avg} of the multithreaded programs shown in Figure 4.11

caused by the first or the last few iterations from the main loop, allowing the main loop to be further optimised and then parallelised. This section focuses on the main parallelisable loop and examines further use of loop peeling.

Figure 4.16 gives the average ratio of the instructions executed (*RIE*) by the master TPU to those executed by the slave TPU. The *RIE* implies how the master TPU is utilised in comparison to the slave TPUs. In the multithreaded execution, each slave TPU may be reused by multiple slave threads, which is called *recycling* execution. From the graph, the master TPU is utilised less than 20% of the average utilisation of a slave TPU. Exceptions are *F_6* and *N_14*. In *F_6*, the multithreaded loop resides in a serial loop (which is executed by the master) and its upper bound is not constant. In *N_14*, the multithreaded loop is followed by a serial loop and they cannot overlap; however, *N_14 (d)* gives the ratios after the number of instructions executed in the serial loop is deducted from the total number of instructions executed by the master.

Due to the nature of the kernel code, while the slaves are executing the loop, little useful computation is left to the master. In the next experiment, early iterations of the loops were peeled prior to the multithreaded transformation. Once transformed, downward code motion is applied to allocate the peeled iterations to the master thread. If there are multiple exits from the loops, *abort cluster* instructions are inserted in the master's code ahead of those exits. The following variations were explored:

- *p.00* represents the original version of the multithreaded loop.
- *p.05* represents the loop in which 5% of the iterations were peeled.
- *p.10* represents the loop in which 10% of the iterations were peeled.
- *p.20* represents the loop in which 20% of the iterations were peeled.

The percentage of iterations peeled is limited to 20% so that its sequential execution does not dominate the overall program execution. Due to the characteristic of the multithreaded loop in *F_6*, as mentioned earlier, it is excluded from the experiment.

The resulting speedup is shown in Figure 4.17, and the *RIE* graphs and their standard deviations are shown in Figures 4.18 and 4.19, respectively.

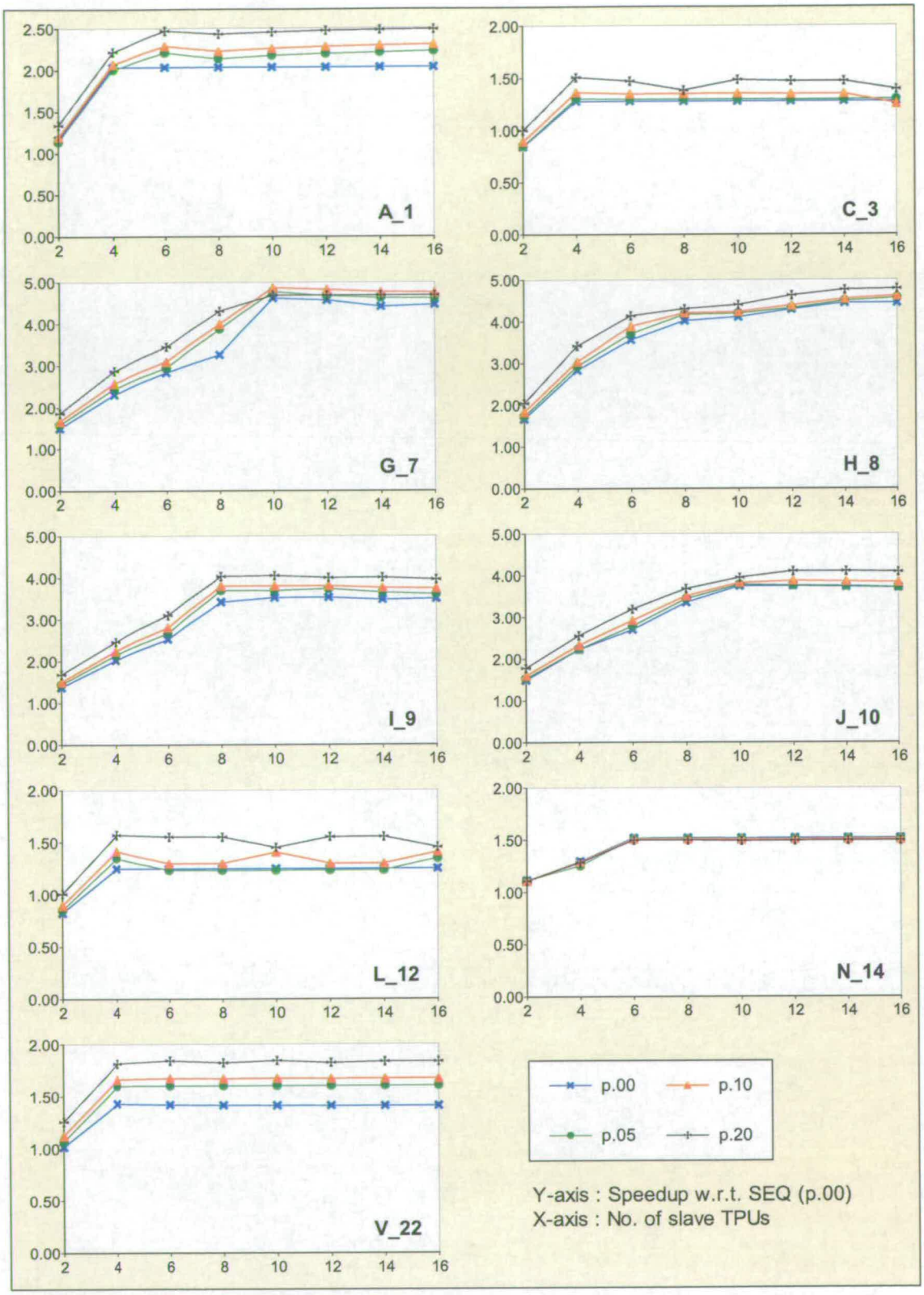


Figure 4.17: Speedup of *recycling* multithreaded execution after loop peeling

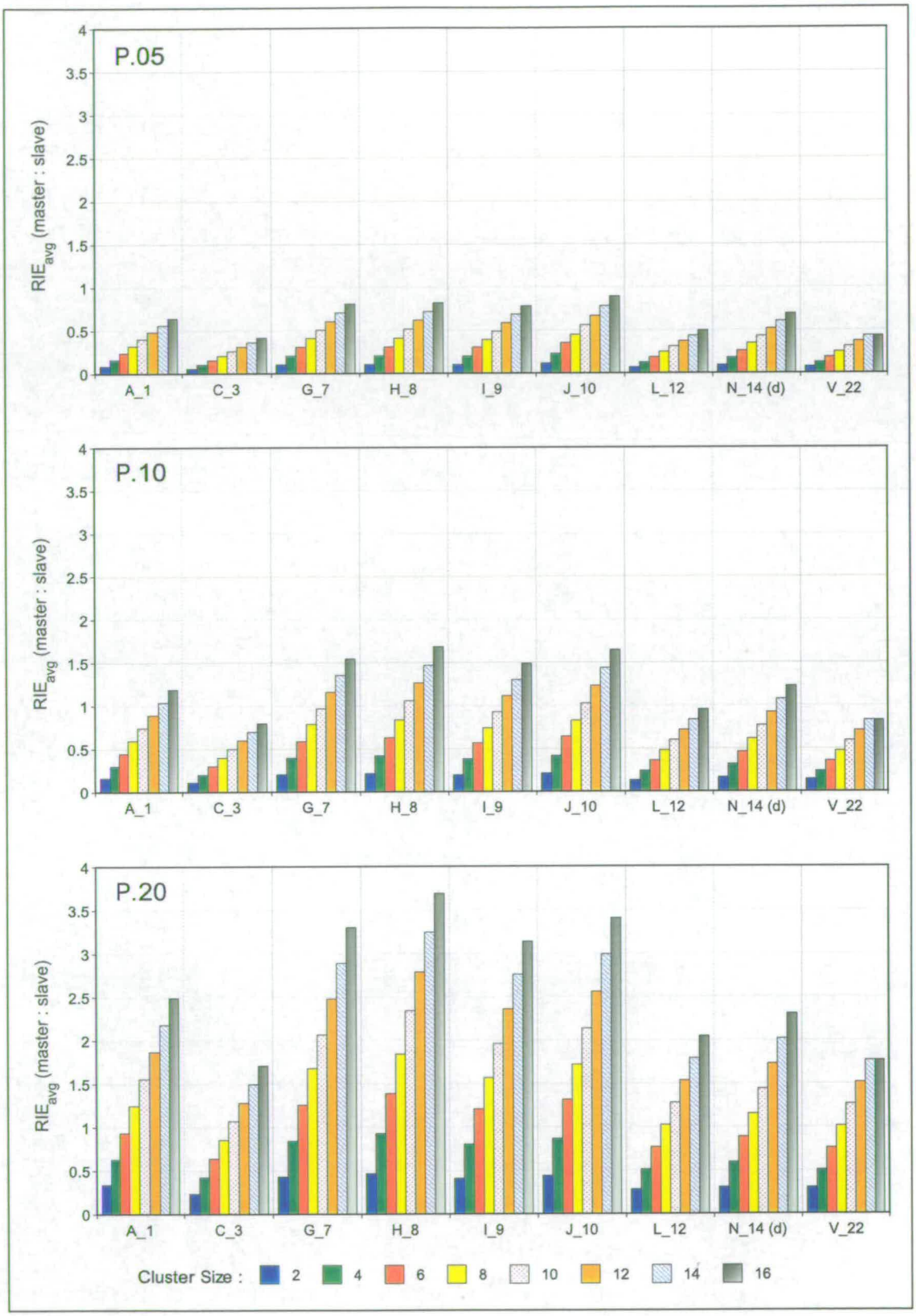


Figure 4.18: RIE_{avg} graphs after loop peeling

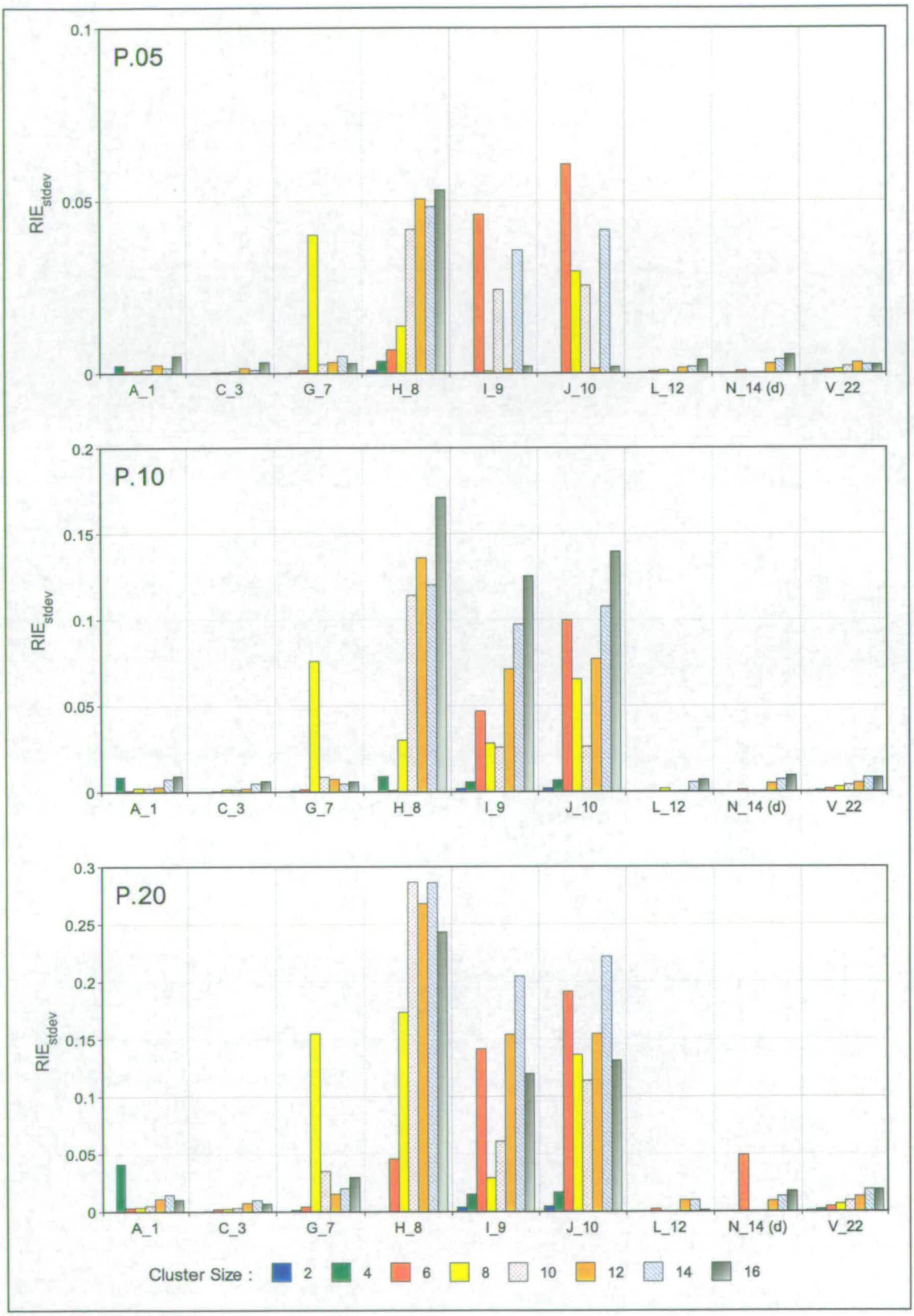


Figure 4.19: Standard deviations of the RIE bars in Figure 4.18

The multithreaded programs with loop peeling increase program speedup with the improvement up to 20%. In *N_14*, there is no improvement at all since the program's speedup is restricted by the serial loop execution. Comparisons of the *RIE* bars in Figures 4.16 and 4.18 reveal that the utilisation of the master TPU is substantially improved. However, as more iterations are allocated to the master, e.g. 10% - 20%, and the cluster size increases, the slave TPUs appear to be under-utilised in comparison to the master TPUs. Consequently, the program performance, in spite of some modest speedup, is limited by the sequential execution of the master thread. The standard deviation of the *RIE*, shown in Figure 4.19, implies how the iterations or threads are distributed among the slave TPUs. When there are fewer threads to distribute to the slave TPUs, i.e. *p.10* and *p.20*, uneven workload becomes more visible, especially in those benchmarks composed of larger threads such as *H_8*, *I_9*, and *J_10*.

4.2.2.2 Loop Unrolling

Loop unrolling replicates the body of loops. If a loop is unrolled n times, then the new loop body contains $n+1$ copies of the original loop body and the iteration step of the new loop is multiplied by $n+1$. It is a common technique to increase the size, and therefore instruction-level parallelism, of the loop body which corresponds to thread size in the multithreaded execution.

First, the impact of loop unrolling on the *recycling* multithreaded execution is studied. The following conditions are explored:

- *b.1* represents the original loop.
- *b.x2*. The loop is unrolled once.
- *b.x4*. The loop is unrolled 3 times.

- *b.x8*. The loop is unrolled 7 times.

5% of the total loop iterations plus leftovers are peeled (they are early iterations of the loop) so that the remainder is an exact multiple of the unrolling factor plus one. Exceptions are *H_8*, *I_9*, and *J_10*. In these benchmarks, the loops have only few iterations and the loop bodies are quite large; thus only the leftovers are peeled so that the master TPU does not execute more (original) loop iterations than those executed by a slave TPU.

Graphs shown in Figures 4.20 and 4.21 demonstrate that a combination of loop unrolling and loop peeling yield significant speedup for most benchmarks. The performance gained in *N_14* is limited by the serial loop execution in the program, whereas the performance gained in *N_14 (d)* is more pronounced as the execution time of the serial loop is deducted from the total execution time (of both the sequential and the multithreaded programs) and therefore the speedup observed is due to the multithreaded execution. The upper bound of the loop in *F_6* is not constant; if the number of iterations is less than the number of copies to be replicated, then the loop will be executed sequentially. Therefore the performance drop in *b.x8* is due to the increasing ratio of the sequential execution to the multithreaded execution. Finally, *H_8*, *I_9*, and *J_10* show little improvement because their original versions already performed well. Moreover, they had few loop iterations. As a result, the more the loops are unrolled, the less is loop-level parallelism exploited.

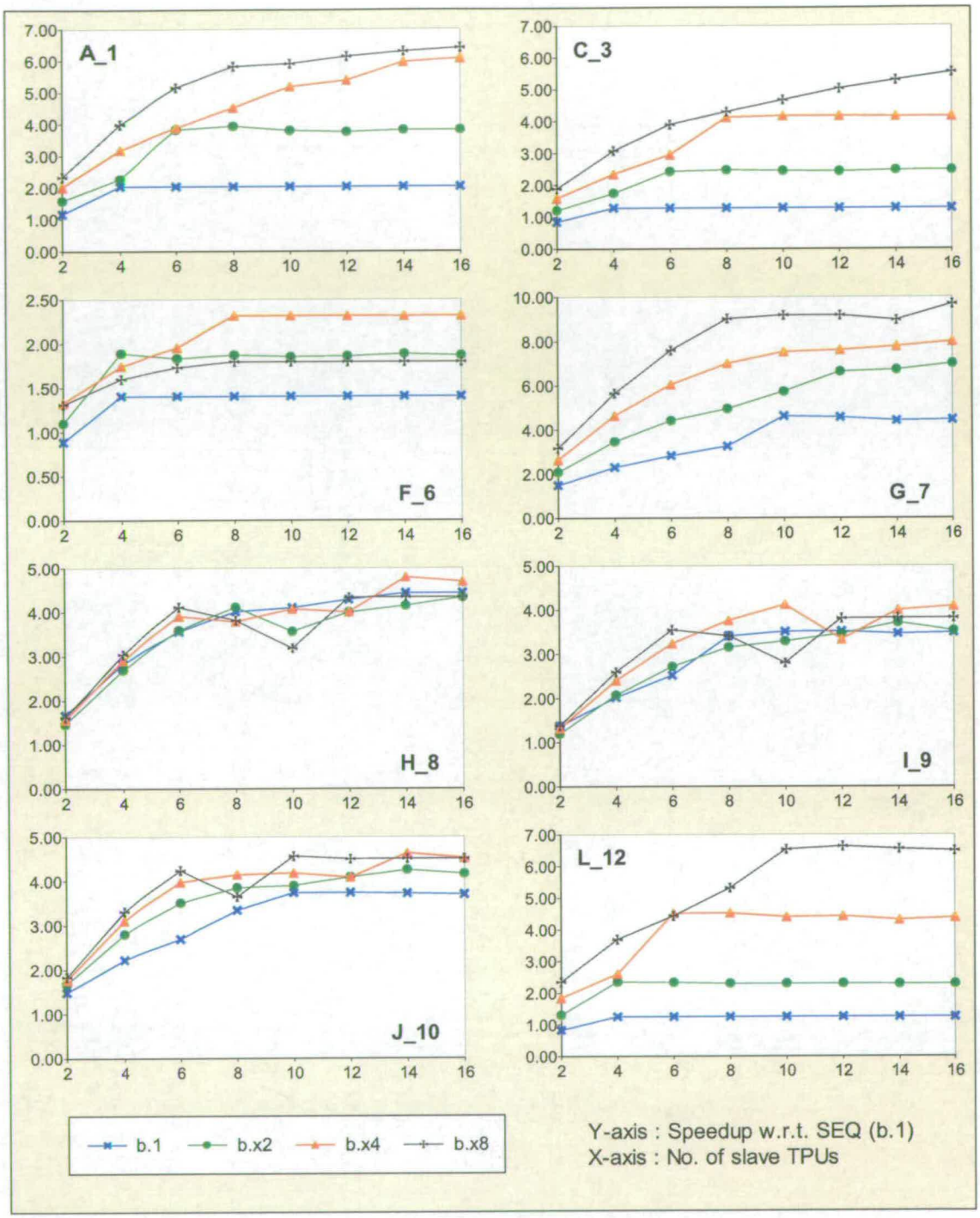


Figure 4.20: Speedup of *recycling* multithreaded execution after loop unrolling and loop peeling (continued in Figure 4.21)

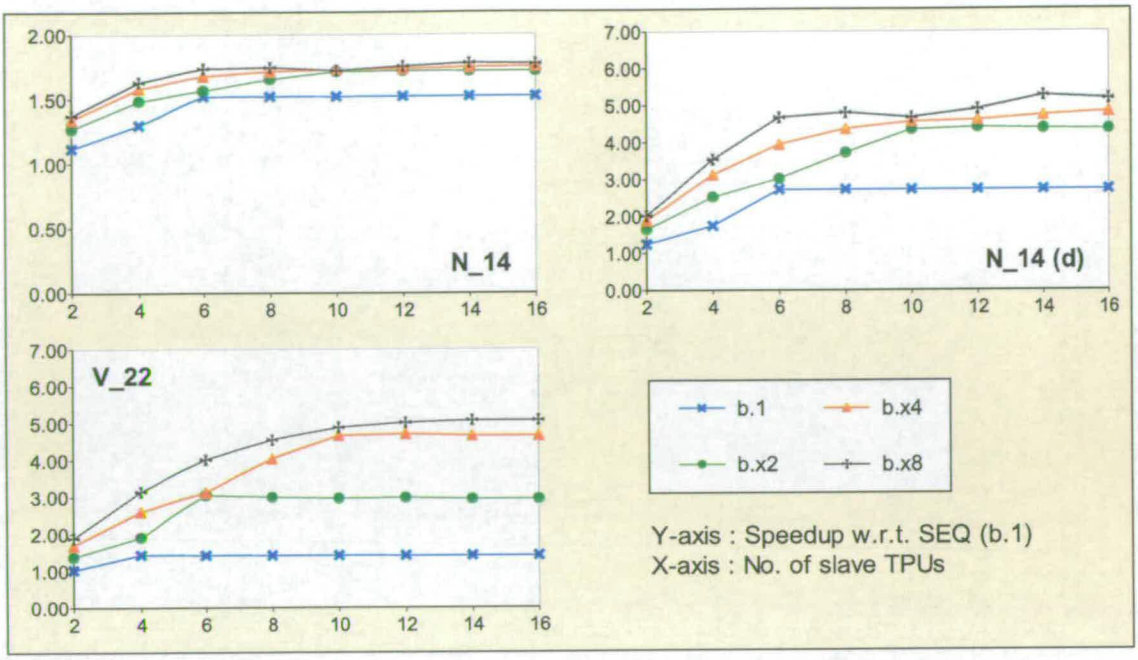


Figure 4.21: Speedup of *recycling* multithreaded execution after loop unrolling and loop peeling (continued from Figure 4.20)

In the *non-recycling* multithreaded execution, multiple threads cannot reuse the slave TPUs. It is more logical to allocate a chunk of iterations to each thread where the size of the chunk has impact on loop-level parallelism. In the second experiment, the benchmarks were optimised to fit resource utilisation of the non-recycling model and avoid any fork failure. For instance, if the loop in *L_12* that comprises 1000 iterations is to be executed by 4 TPUs, it will be unrolled 249 times to generate 4 chunks of 250 iterations at the most. These chunks are re-rolled, producing small loops similar to the original one but with conditional exit and with adjusted upper and lower bounds[1], as illustrated in Figure 4.22. This was called *loop chunking* by Olukotun et al. [53]. The maximum number of iterations per chunk is $\left\lceil \frac{\textit{the total no. of iterations}}{\textit{the total no. of TPUs}} \right\rceil$. Then, a (full)

[1] In practice, the loop is never unrolled and re-rolled - the new loops are constructed by modifying the original one.

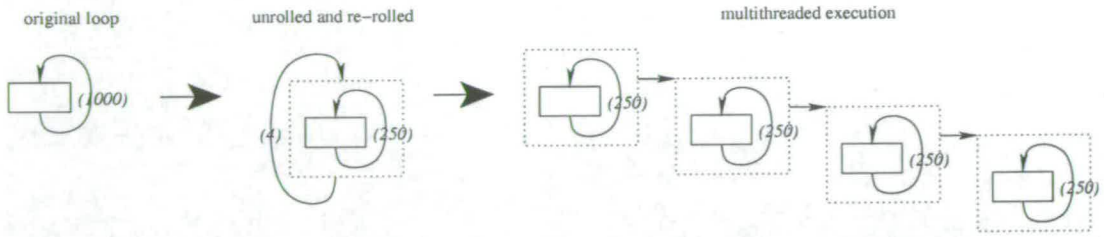


Figure 4.22: Loop chunking for multithreaded execution on 4 TPUs

chunk and leftover iterations can be jammed, peeled, and allocated to the master TPU while the rest are distributed among the slaves.

The optimisation is performed prior to the multithreaded transformation. The total number of TPUs in the experiment, including the master and the slaves, is varied from 2 to 16. *F_6* is excluded from the experiment because the number of iterations of the multithreaded loop is not constant and is unknown at the compile-time - the compiler would let *F_6* pass without any modification.

The results are shown in Figure 4.23. Reasonable speedup can be seen in all the benchmarks since each of their multithreaded version is specifically compiled to match the number of TPUs available in the cluster. Because each TPU hosts only one thread that performs its computation in parallel with the others, the execution time of a loop is approximately the average execution time per thread, which corresponds to the amount of computation in a chunk, plus the total delays between all the threads. When the number of TPUs increases to a point that the chunk becomes too fine or there are insufficient iterations to allocate to every TPU, then the program performance will no longer improve. An example is *H_8*, in which the number of iterations per chunk when there are {2, 4, 6, 8, 10, 12} TPUs participating in the execution are {50, 25, 17, 13, 10, 9}, respectively. The speedup significantly rises when 2-6 TPUs are used as the

chunks size is reduced from 50 to 17 iterations. It then levels off when more than 6 TPUs are used as the chunk size is almost unchanged. However, when the number of TPUs increases to 14 (13 slaves plus a master), only 12 slave TPUs are actually used because there are not enough iterations to allocate to the last one.

Comparing these results to the ones from the recycling model (Figures 4.20 and 4.21), particularly *b.x8*, demonstrates that both versions give fairly similar speedup. Exceptions are *G_7* and *L_12*, where the recycling model with loop unrolling performs noticeably better than the non-recycling one with loop chunking. In the former, new unrolled iterations can be further optimised during the back-end compilation. For example, some memory references are replaced by registers and repetitive address calculations are eliminated. In the latter, the new loop iterations are rolled back and even more instructions are added for checking and adjusting the loop bounds. Therefore, the performance gained from applying loop chunking to the non-recycling execution is due to the fact that the iterations are allocated to match the availability of resources, thus eliminating the fork penalty. However, there is still the overhead of the chunking added to each thread.

In the next experiment, loop chunking was applied in conjunction with nested multithreading in benchmarks *R_18* and *U_21*. The multithreaded execution in nested loops is non-recycling. The benchmarks were prepared as described next.

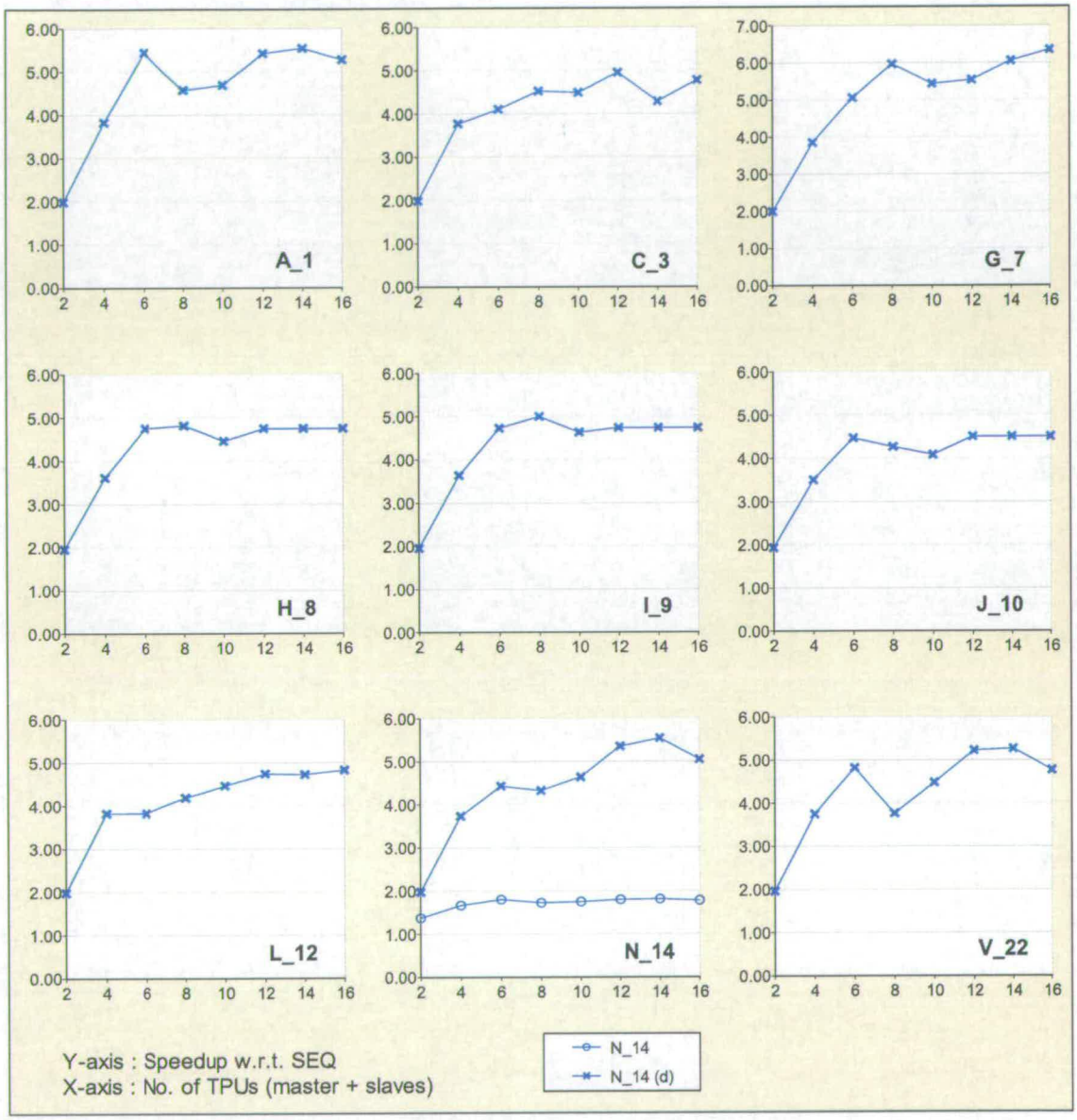


Figure 4.23: Speedup of *non-recycling* multithreaded execution after loop chunking

- For *R_18*,
  - {*N(2,2)*, ..., *N(2,7)*} allocate 2 slave TPUs to the outermost loops while the innermost loops were chunked, with the number of TPUs ranging from 2 to 7.
  - {*N(3,2)*, ..., *N(3,4)*} allocate 3 slave TPUs to the outermost loops while the innermost loops were chunked, with the number of TPUs ranging from 2 to 4.
- For *U_21*,
  - The innermost loops were always executed sequentially.
  - {*N(2,2,1)*, ..., *N(2,6,1)*} allocate 2 slave TPUs to the outermost loops while the middle loop was chunked, with the number of TPUs ranging from 2 to 6.
  - {*N(3,2,1)*, ..., *N(3,4,1)*} allocate 3 slave TPUs to the outermost loops while the middle loop was chunked, with the number of TPUs ranging from 2 to 4.

Figure 4.24 compares the results from before (Figures 4.13 and 4.14) and after the optimisation. Fair improvement can be seen in both the benchmarks, with the increase in the speedup ranging between 25% and 30%. In the case of *U_21*, the benefit of the optimisation is less pronounced as more TPUs are used to execute the middle loop. This is due to the fact that it comprises only 25 iterations, and the amount of work per thread when more than 3 TPUs are used is little different, i.e. the number of iterations executed by a thread when there are $\{2, 3, 4, 5, 6\}$ TPUs participating in the execution are $\{13, 8, 6, 5, 4\}$, respectively.

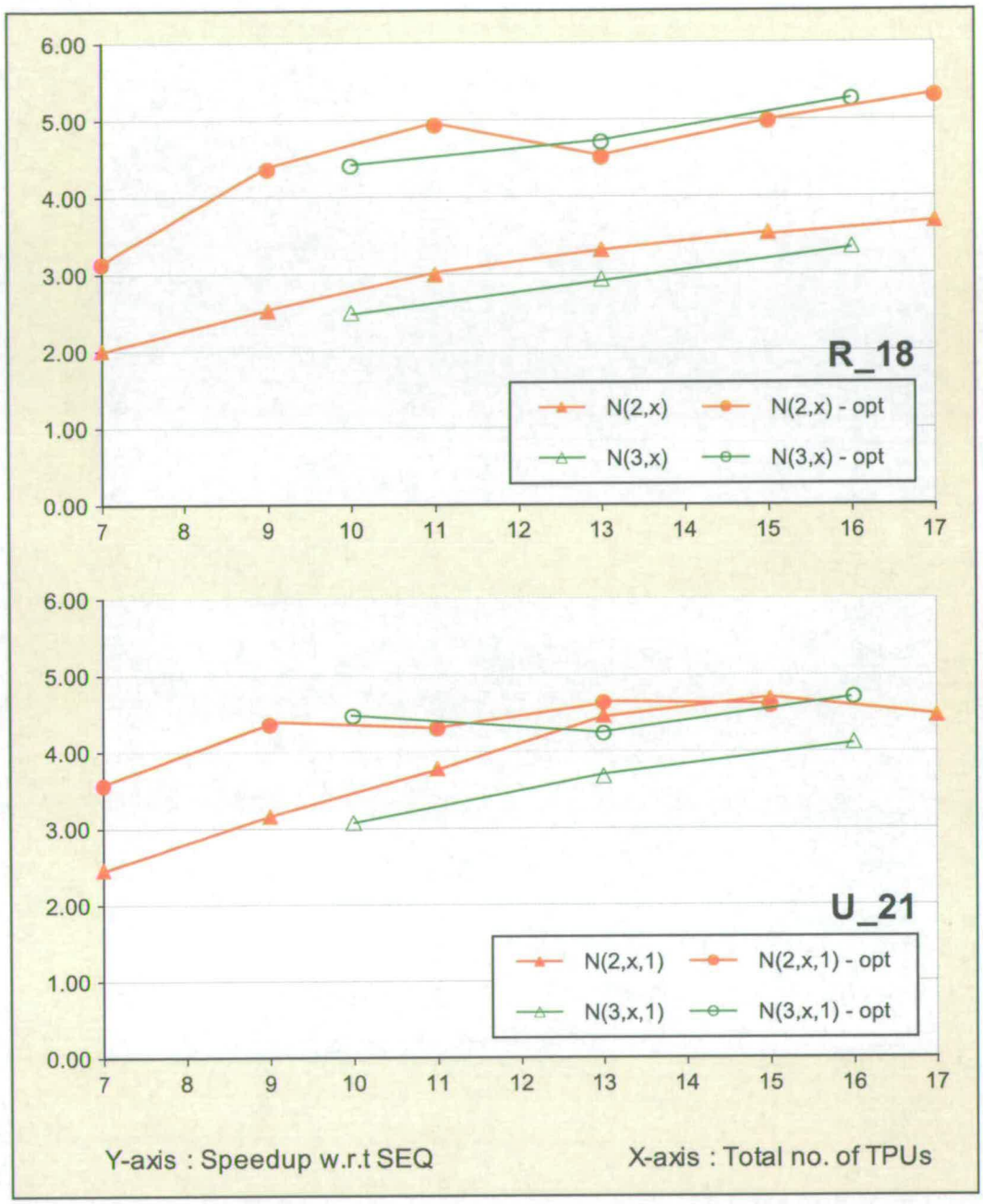


Figure 4.24: Speedup of nested-multithreaded programs with and without optimisation (loop chunking)

4.2.2.3 Cluster and Fork Penalties

According to the multithreaded execution model, if the master thread fails to form a slave cluster, then it has to execute the whole loop by itself. Although some instructions whose guard values are zeros can be bypassed, the performance of the transformed loop executed sequentially may still be worse than the original loop's. Once a cluster is allocated, if a thread fails to fork a new slave, its penalty is to execute the next iteration instead of retiring. Two multithreaded versions of each benchmark were prepared:

- *M_SEQ* is the multithreaded program using cluster size 2.
- *S_SEQ* is the multithreaded program using cluster size 1.

The total number of TPUs in the architecture is changed to 2. Because of this, `cform` operations in *M_SEQ* always fail. On the contrary, those in *S_SEQ* always succeed, although the sole thread in the cluster always fails in `xfrk`. Hence both *M_SEQ* and *S_SEQ* are always executed sequentially.

The performance displayed in Figure 4.25 indicates a worst case of the cluster and fork penalties in the unoptimised multithreaded programs. Given the performance lost from 100% of the cluster and fork failures, an average speedup of all benchmarks is around 0.8. In *C_3*, *F_6*, and *L_12*, the speedup is only around 0.6 as they contain very small loops. Optimistically, with a combination of loop unrolling and loop peeling such as the *b.x8* strategy (in Figures 4.20 and 4.21), the speedup of the recycling multithreaded execution could be around 5 or higher if the cluster and fork operations succeed, or closer to 1 if they all fail.

Figure 4.25: Speedup of multithreaded programs being sequentially executed

Loop chunking restructures the loop prior to the multithreaded transformation, which allows the number of threads created to match the number of TPUs available. However, in the next experiment, a loop is restructured so that n_1 chunks are created, but it is then transformed to be executed by n_2 TPUs, $n_2 \leq n_1$. The benchmarks were prepared as follows:

- *cmpl.T6.* The loop is restructured to create 6 chunks, and multithreaded transformed using cluster size $\{1, 3, 5\}$.
- *cmpl.T8.* The loop is restructured to create 8 chunks, and multithreaded transformed using cluster size $\{1, 3, 5, 7\}$.
- *cmpl.T10.* The loop is restructured to create 10 chunks, and multithreaded transformed using cluster size $\{1, 3, 5, 7, 9\}$.

The total number of TPUs actually used is equal to cluster size plus one. In *cmpl.Tmatch*, the loop is restructured and transformed such that the number of threads created is equal to the number of TPUs, i.e. {2, 4, 6, 8, 10, 12, 14, 16}. They are the same programs as the ones shown in Figure 4.23.

In Figure 4.26, the performance of *cmpl.T6*, *cmpl.T8*, and *cmpl.T10* slightly drops when there are more threads than the TPUs available. A common observation in all benchmarks is that although both *cmpl.T8* and *cmpl.T10* suffer from fork penalty when they are given 6 TPUs, *cmpl.T10* always performs better than *cmpl.T8*. This can be explained by the fact that *cmpl.T10* generates more threads, thus exposing more loop-level parallelism when the slave TPUs are reusable. The loops in this experiment are un-nested, which allows the masters to signal the slaves immediately after completing their execution. Even in nested loops, the execution can switch between recycling and non-recycling; this depends on the length of the outer and the inner loop bodies and the passing of the synchronisation signals at run-time. Therefore, the fork penalty observed in this experiment is optimistic.

cmpl.Tmatch gives an upper bound of the multithreaded performance. Its approach involves restructuring and multithreading a loop for every specific number of TPUs. The loop that is restructured so that too few chunks are created offers little flexibility to the multithreaded transformation and execution. Hence, an optimistic approach should allow the pre-processor to create more chunks than the number of TPUs estimated by the multithreaded transformer to be available. For example, from Figure 4.26, it may be worth using the *cmpl.T10* strategy if the loop can be pre-processed only once because the disadvantage of this program being executed by fewer than 10 TPUs is not too severe, considering its speedup and the best one achieved by *cmpl.Tmatch*.

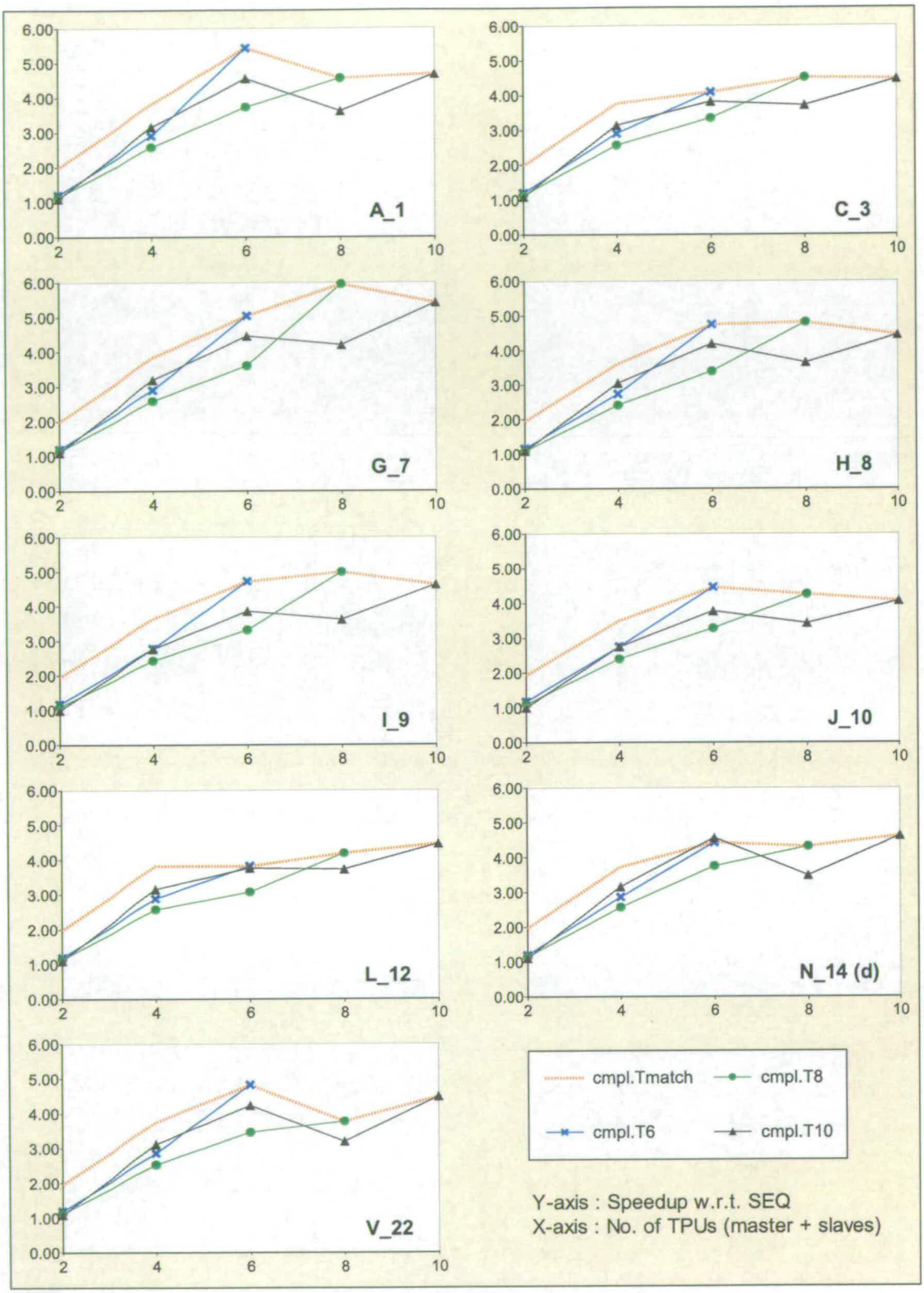


Figure 4.26: Speedup of multithreaded programs with fork penalty

4.2.3 Summary

The following conclusions can be drawn from the experiments. Firstly, because our multithreaded execution relies on the software thread manipulation, the thread size should be large enough so that the benefit gained from the multithreading outweighs the overheads. Loop unrolling is employed for this very purpose. Furthermore, the resource utilisation of the master TPU can be improved with the application of loop peeling. The combination of both techniques achieved speedup between 5-10, when the loops were unrolled 7 times and one-level multithreading was applied. Figures 4.27 and 4.28 summarise the performance of the one-level multithreaded programs.

Limitation of multithreading in nested loops was noted. There are several clusters executing outer and inner loops simultaneously at different nest levels. In the current system, a unique (synchronisation) signal can be received by one thread at a time, which allows the thread to commit, retire, and free the TPU. As a result, while the signal is passed around in one cluster, the others repeatedly suffer from fork failures since the TPUs cannot be recycled. Aggressive loop unrolling provides a solution to this. Chunks of iterations are generated to match the number of TPUs available and allocated to individual threads. Speedup between 4-5 was achieved after the restructuring of the inner loops in the nests (as seen in Figure 4.24). However, this technique compromises loop-level parallelism if too few threads are created while the TPUs are reusable at run-time.

From the hardware perspective, increasing the number of TPUs allows more overlapping computation. However, there are points after which the increase in the number of TPUs will no longer improve the program performance. A case for the recycling execution is: when the execution pattern of the loop reaches a point that a new thread can reuse a TPU which has been freed by a previous thread, instead of using a new one.

For the non-recycling execution, which assigns a chunk of iterations to an individual thread, the chunk size is reversely proportional to the number of TPUs participating in the execution. If the chunk is already small, then adding an extra TPU will result in even finer threads, which is no further beneficial to the multithreaded execution.

There are other compiler techniques which have not been explored. Because data cache is omitted from the simulation, techiques such as strip-mining or loop tiling which improve memory locality were not considered. In addition, most benchmarks contain small single loops, providing no opportunity for the application of loop fusion or loop fission. Finally, loop coalescing and loop collapsing which transform nested loops into single-level ones were not considered as the multithreaded execution in nested loops is one of the subjects to be examined in this research.

4.3 Chapter Summary

Two multithreaded loop transformers were implemented using SUIF framework. One handles simple loops with only natural exits. The other handles loops with multiple exits or whose upper bounds are unknown; such cases require the execution to be speculative. Results from the preliminary experiments were reported and discussed. Generally, the multithreaded programs deliver reasonable speedup with respect to the sequential ones. Other traditional techniques such as loop unrolling and loop peeling were also applied to improve the multithreaded performance.

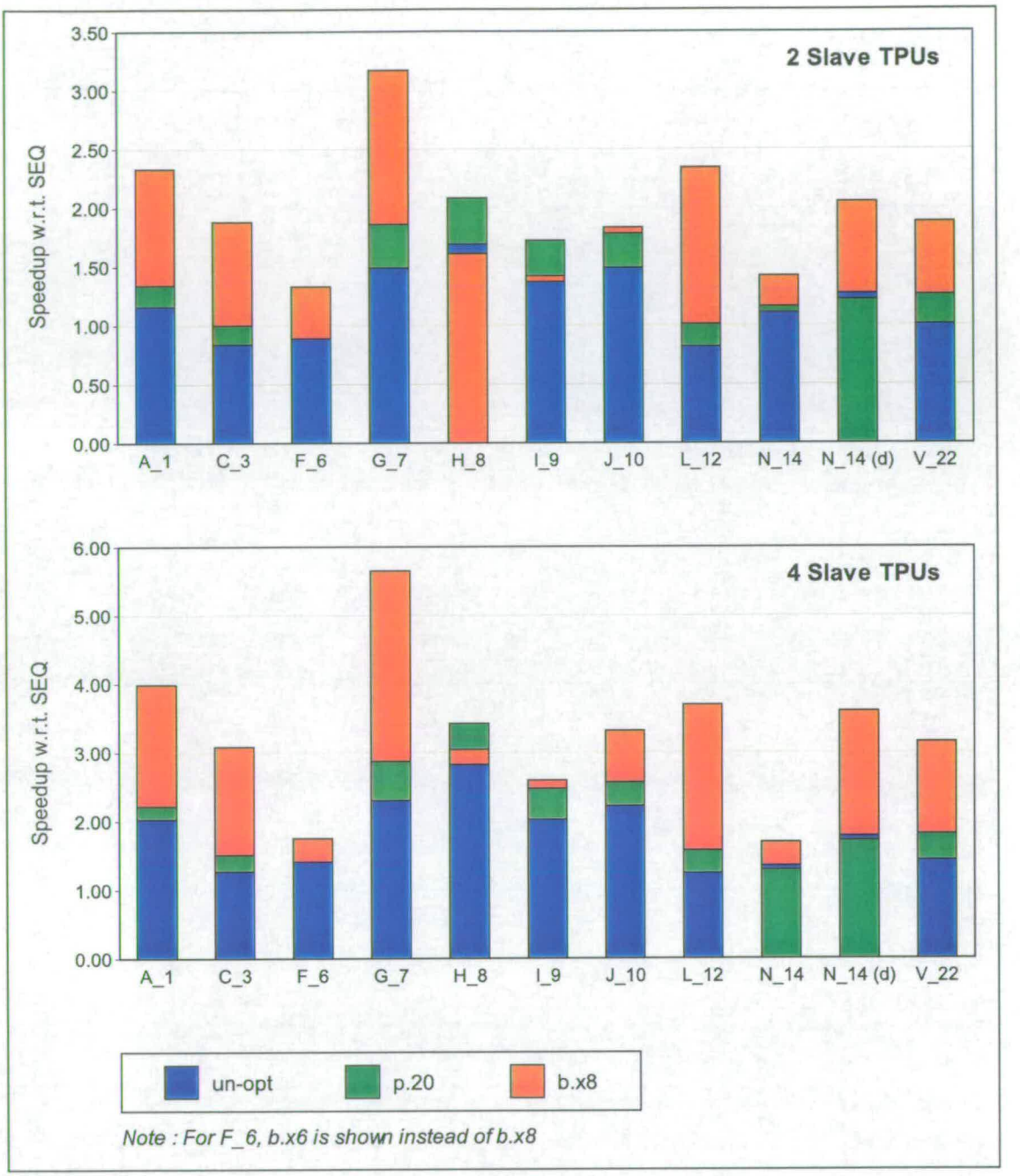


Figure 4.27: Performance of one-level multithreaded programs (continued in Figure 4.28)

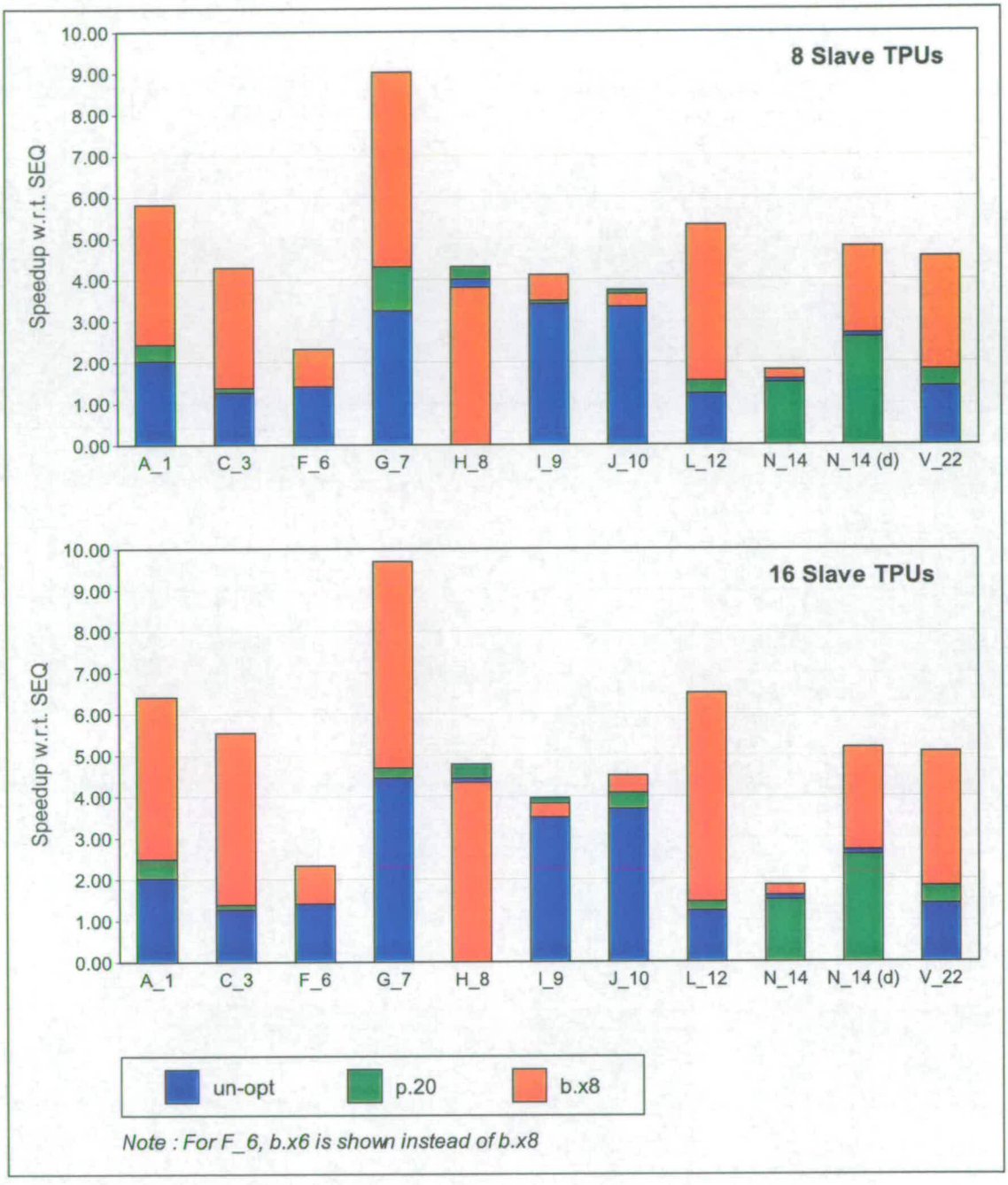


Figure 4.28: Performance of one-level multithreaded programs (continued from Figure 4.27)

Chapter 5

Multithreaded Control-Speculative Execution

Control-speculative execution permits either or both control-dependent paths of a branch to be executed before the outcome of that branch is known. In the multithreaded execution, the speculated paths are typically executed by separate threads. The choice of which path to speculate on is made using *profile-based branch prediction*. The studies in [13, 24] revealed that most branches, especially the ones that can take either direction with high probability, exhibit the same behaviour across different program executions that use different input data. Strategies which rely on static program analyses were studied by [8, 64]. Their findings were that a branch, that chooses between continuing or exiting a loop or a procedure, is likely to take the continuation path. Moreover, the path that does not contain function calls is more likely to be taken since most programs use conditional calls to handle exceptions which rarely occur.

If a branch has low confidence, i.e. both paths are equally probable, then the speculation may be omitted; alternatively, by employing *dual-path speculation* both threads

are launched to execute speculatively. For branches with high prediction confidence, *single-path speculation* forks only one thread to execute the more probable path. Deciding whether a branch has low prediction confidence based on the difference in probabilities is subjective. It depends on a number of different factors such as the prediction accuracy, the misprediction penalty, or the resource availability.

Control speculation allows several program partitions to be executed simultaneously, each of which may, in turn, be executed by multiple threads. Empirical studies undertaken as part of this work prioritised concurrent program partitions and evaluated resource allocation strategies. In the next section, the transformation for multithreaded control-speculative execution is first explained.

5.1 Transformations for Control Speculation

The transformers process SUIF programs in which high-level TREE_IF nodes are marked and dismantled into straight-line code, and low-level branch instructions are recognised[1]. The following analysis are performed prior to the transformation.

The program is compiled procedure-by-procedure, for each one, a control-flow graph (CFG) is constructed. The first node in the graph is always *ENTRY* and the last one is either *EXIT* or *RETURN*, as shown in Figure 5.1. Dominator (or pre-dominator) and post-dominator nodes of the branches are calculated [2, 4], which is described next. Given two nodes *n1* and *n2* in a CFG, *n1* dominates *n2* if every path from *ENTRY* to *n2* goes through *n1*. Similarly, *n2* post-dominates *n1* if every path from *n1* to *EXIT* goes through *n2*. Based on these definitions, the control-flow from *n1* to *n2* is considered

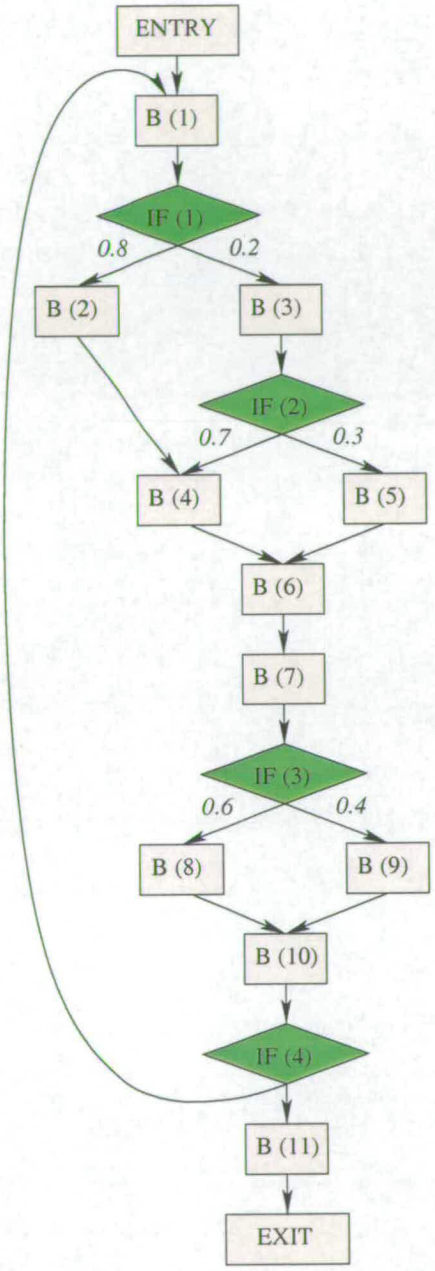
[1] A dismantled TREE_IF is arranged such that the original THEN path becomes the fall-through path and the branch is instead to the original ELSE path. To maintain consistency with other low-level structures, the fall-through path is called the ELSE path and the target of the branch is always the THEN path.

backward if *n2* dominates *n1*, and a branch is considered a *forward branch* if it is not dominated by either of its targets.

For each forward branch, parent and child regions represent boundaries at which speculation might be applied. The parent region is constructed by traversing the CFG upward, starting from the branch. Its dominators are added to the region until the first node which is not a dominator is reached or the re-convergent node of the previous branch has been included. On the other hand, traversing the CFG downward, starting from the branch, two child regions include nodes along THEN and ELSE paths. The construction of each region stops when the first post-dominator or the re-convergent node of that branch is reached. If a branch is found in a child region of another branch, then the parent region of the embedded branch is truncated so that only nodes in the enclosing child region are included.

An example is shown in Figure 5.1. Node *B* represents a block of instructions as in a basic block, but the branch instruction at the end of the block is represented separately as an *IF* node[2]. The edge from *IF(4)* to *B(1)* represents backward control-flow. Forward branches are *IF(1)*, *IF(2)*, and *IF(3)*. Their dominators, post-dominators, parent regions, and child regions, are calculated as shown in the figure. As *IF(2)* is located on a control-dependent path of *IF(1)*, the parent region of *IF(2)* is truncated so that it is within the child region of *IF(1)*.

[2]In subsequent graphs in this thesis, only forward branches are represented separately.



Forward Branches :

IF (1)

Pre–dominators	{ B(1) }
Post–dominators	{ B(6), B(7), IF(3), B(10), IF(4), B(11) }
Parent Region	{ B(1) }
Child Region 1	{ B(2), B(4) }
Child Region 2	{ B(3), IF(2), B(4), B(5) }

IF (2)

Pre–dominators	{ B(1), IF(1), B(3) }
Post–dominators	{ B(6), B(7), IF(3), B(10), IF(4), B(11) }
Parent Region	{ B(3) }
Child Region 1	{ B(4) }
Child Region 2	{ B(5) }

IF (3)

Pre–dominators	{ B(1), IF(1), B(6), B(7) }
Post–dominators	{ B(10), IF(4), B(11) }
Parent Region	{ B(6), B(7) }
Child Region 1	{ B(8) }
Child Region 2	{ B(9) }

Figure 5.1: An example of a control-flow graph

Table 5.1 Overheads of multithreaded speculative execution

Overheads	Average time units
Spec_Transformer_1 : parent / child	30 / 44
Spec_Transformer_2 : parent / child	40 / 36
Spec_Transformer_3 : parent / child (per branch)	38 / 52

Branch probability is collected from sequential execution profiling and is added to SUIF files by `tcovsuif`. It gives two types of information:

1. The cumulative probability along a control-flow path until a branch is encountered indicates whether that branch significantly contributes to the overall program execution. In the example in Figure 5.1, the cumulative probability of *IF(1)* and *IF(3)* are 1.0, but the cumulative probability of *IF(2)* is only 0.2.

2. The individual probability determines which direction to speculate. Both paths of the branch can be speculatively executed if they are equally probable.

Branches that are too fine for speculation are merged into parent or child regions of their neighbours, if possible. Criteria which are used to determine whether a branch is too fine or not include the cumulative probability of that branch and the size of its parent and child regions. The latter is compared with the speculation overheads in Table 5.1, which are the average execution time of the thread manipulation routines in the parent and the child threads (measured from the experiments in Section 5.2).

For a predicted branch, incoming control-flow from nodes other than itself to the child regions are diverted to new targets, by means of code replication which is similar to tail duplication techniques in superblock or trace scheduling [10, 16, 19]. Outgoing control-flow from nodes other than the last one in the region is permitted from the

parent, but not from the child as the speculative execution can only be performed within the child region's boundary.

In Figure 5.1, the child regions of *IF(1)* and *IF(2)* are overlapped, starting from node *B(4)*. Hence, *B(4)* is replicated and the control-flow from *B(2)* is directed to a new node *B(4')*, as shown in Figure 5.2. The replication of *B(6)* is optional since *B(6)* post-dominates both *IF(1)* and *IF(2)*, but it is not included in the child region of neither branch. However, by adding *B(6')* to the major path of *IF(1)*, the size of the speculated region can be increased to amortise the speculation overheads. The control-flow from unconditional branch or jump instructions are handled in the same way. More examples of code replication can be found in [51].

The final analysis involves extracting data dependency information from each pair of parent-and-child regions. Then, each predicted branch is transformed for the multithreaded control-speculative execution. An overview of the compilation flow is displayed in Figure 5.3.

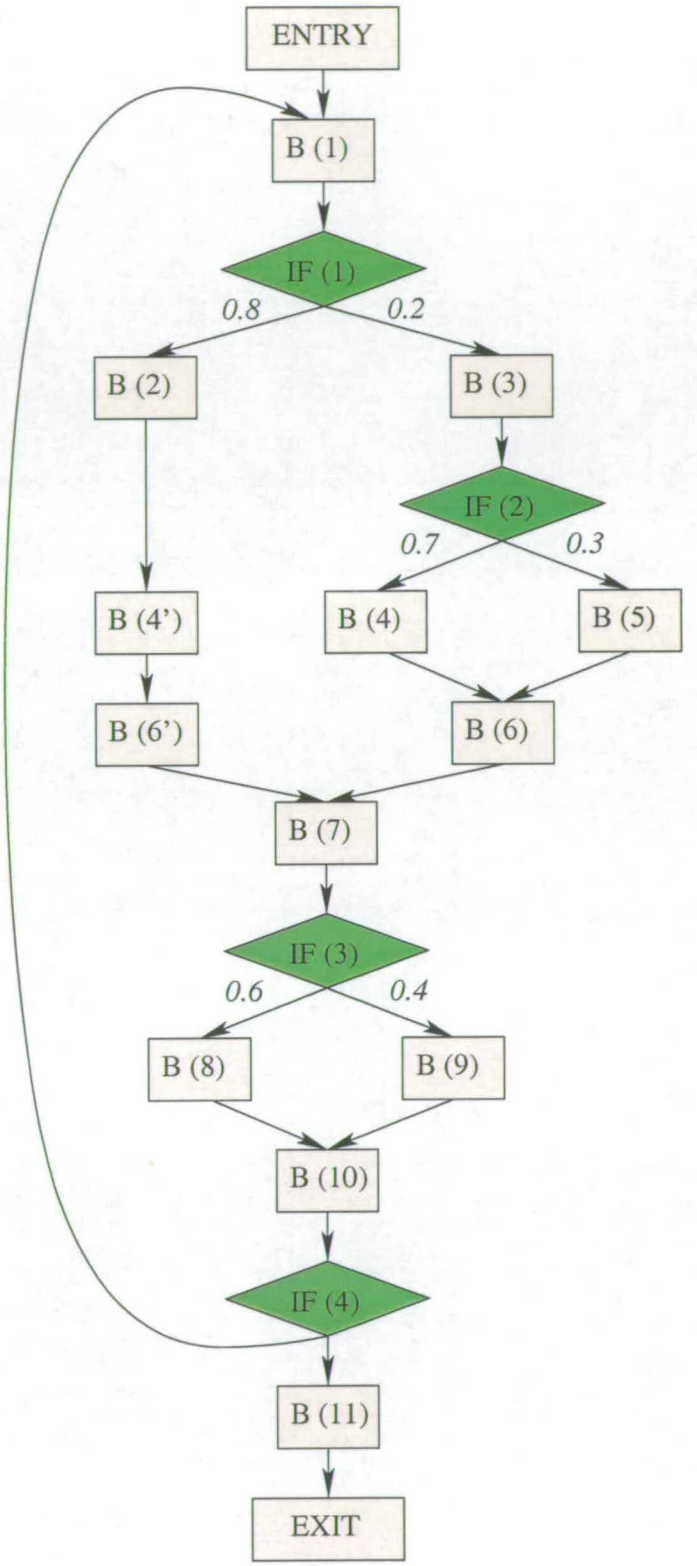


Figure 5.2: The control-flow graph in Figure 5.1 after code replication

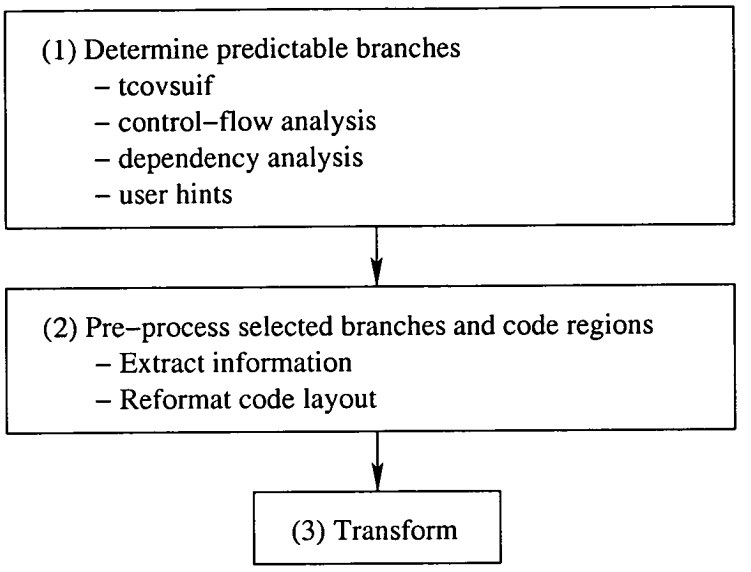


Figure 5.3: An outline of the transformation for speculative execution

5.1.1 Single-Path Speculation

In single-path speculation, a thread is forked to speculatively execute *the predicted path* of the branch. For instance, path { *B(2), B(4'), B(6')* } of the branch *IF(1)* in Figure 5.2 is chosen for the single-path speculation.

The branch structure which has been reformatted for the transformation is shown in Figure 5.4(b), from its original form in Figure 5.4(a). Figure 5.5 gives an example of the transformed code when the THEN path is predicted. Only the lines marked with an asterisk (*) are modified should the ELSE path be predicted; the alternative code can be found in the comment section of those lines.

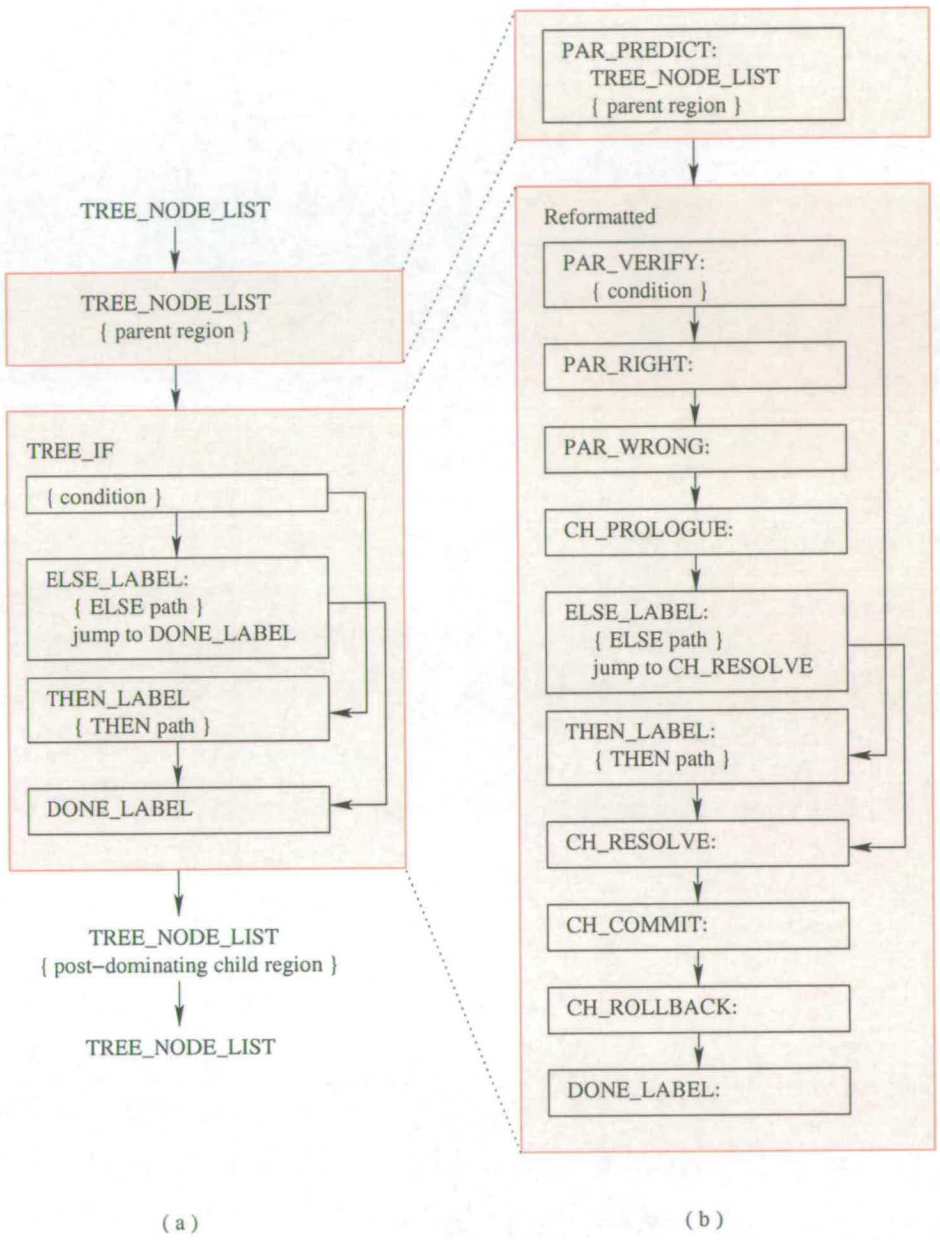


Figure 5.4: Branch structure in the SUIF intermediate representation

```
 1   int guard, fsucc, pbra;                  // working variables
 2   int myself, mychild, myparent;           // working variables
 3   PAR_PREDICT:                             //
 4       fsucc = frk (sequence_no, CH_PROLOGUE); // fork a speculative thread
 5       mychild = cadr ();                   // get child's address
 6       guard = 0;                           // indicate that this is parent thread
 7       ... parent region's code ...         //
 8   PAR_VERIFY:                              //
 9       // branch THEN_LABEL                 // original branch instruction
*10      branch PAR_RIGHT;                    // ''branch PAR_WRONG'' if ELSE is predicted
*11      goto PAR_WRONG;                      // ''goto PAR_RIGHT'' if ELSE is predicted
 12  PAR_RIGHT:                               //
 13      if (fsucc) {                         //
 14          psg (fsucc, mychild, sequence_no); // pass signal to child
 15          sstp (fsucc, mychild);           // parent synchronises and stops
 16      }                                    //
*17      else goto THEN_LABEL                 // ''goto ELSE_LABEL'' if ELSE is predicted
 18  PAR_WRONG:                               //
 19      isg (fsucc, mychild, CH_ROLLBACK);   // interrupt child's execution
*20      goto ELSE_LABEL;                     // ''goto THEN_LABEL'' if ELSE is predicted
 21  CH_PROLOGUE:                             //
 22      guard = 1;                           // indicate that this is child thread
 23      safe (guard, 0);                     // become speculative (unsafe)
 24      myparent = padr ();                  // get parent address
*25      goto THEN_LABEL;                     // ''goto ELSE_LABEL'' if ELSE is predicted
 26  ELSE_LABEL:                              //
 27      pbra = 1;                            // post-dominating instructions excluded
 28      ... ELSE path's code ...             //
 29      goto CH_RESOLVE;                     //
 30  THEN_LABEL:                              // predicted path
 31      pbra = 0;                            // post-dominating instructions included
 32      ... THEN path's code ...             //
 33  CH_RESOLVE:                              //
 34      if (!guard) goto DONE_LABEL;         // if this is parent, exit
 35      wat (guard, sequence_no);            // if this is child, wait for the signal
 36  CH_COMMIT:                               //
 37      cmmt (guard);                        // commit speculative stores
 38      goto DONE_LABEL;                     //
 39  CH_ROLLBACK:                             //
 40      ... abort slaves in THEN ...         // or ELSE if it is predicted
 41      stp (guard, -1);                     // child stops
 42  DONE_LABEL:                              // exit
 43  if (pbra)                                //
 44      { ... post-dominating region ... }   //
```

Figure 5.5: Code generated by **Spec_Transformer_1**, THEN path is predicted

Parent Thread

The parent thread speculatively forks a child (line 4) before continuing its execution in the parent region. As the target of the predicted branch is always `THEN_LABEL` (line 9), it is verified as follows:

1. If the THEN path is predicted:
    - the prediction is correct if the branch is taken (`branch PAR_RIGHT`).
    - the prediction is wrong otherwise (`goto PAR_WRONG`).
2. If the ELSE path is predicted:
    - the prediction is wrong if the branch is taken (`branch PAR_WRONG`).
    - the prediction is correct otherwise (`goto PAR_RIGHT`).

At `PAR_RIGHT`, if the `frk` instruction had succeeded, then the parent thread passes a signal to its child (line 14) and stops (line 15). The parent retires only when it becomes the *head* thread; therefore, `sstp` instruction is used. If the `frk` had failed, then the parent has to execute the correct path itself (line 17) since no child thread had been spawned. In case of a wrong prediction, i.e. `PAR_WRONG`, the parent interrupts its child's execution (line 19), and goes to the correct path (line 20).

Child Thread

The child's execution starts at `CH_PROLOGUE` (line 21). Being a speculative thread, it turns off the *safe* flag (line 23) before jumping to the predicted path (line 25). After the path's execution, it waits for a signal from its parent (line 35). If the prediction is correct, then all the speculative stores will be committed (line 37) as soon as the signal

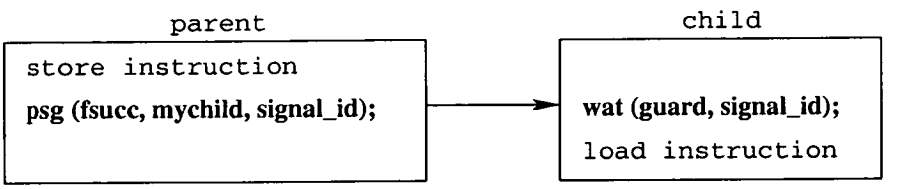


Figure 5.6: Memory communication in **Spec_Transformer_1**

is received. The child will also be appointed the next *head* thread and leave the branch structure at DONE_LABEL.

There can be multithreaded loops along the speculated path. For a series of them, only the first loop is actually a speculative one since the master thread is always blocked when trying to pass the synchronisation signal to the slaves. Once the branch prediction is verified and the loop is on the correct path, the execution can resume and move on to the next loop. If the prediction is wrong, the child will be unblocked or interrupted from its current execution. It then jumps to CH_ROLLBACK and aborts the slave cluster before stopping.

Data Dependence

The data dependence between parent and child threads is handled in the same way as in the loop transformations. Memory communication for each dependent instruction pair is shown in Figure 5.6. Synchronisation between the parent and the child is enforced by passing and waiting for a signal. In the case of register communication (see Section 4.1.3), a set of registers that may cause data dependencies is declared by `uregs` instruction which is inserted before the `frk` (line 4). These registers can be forwarded to the child thread by `fregs` instruction, once they are available.

For a large number of data dependencies, the register communication would in practice be less costly than the memory communication. This is due to the following two reasons: no extra instruction is needed for the child thread, and the parent can forward up to 32 registers in one instruction. However, since the register usage information is required, this has to be done during the back-end compilation, possibly in conjunction with the instruction scheduling, and should ensure that the child thread does not starve of registers.

Post-Dominating Region

To increase the thread size, post-dominating instructions, or instructions below the re-convergent point, might be included in the child thread. A branch structure is *symmetric* if both paths exclude the post-dominating code, or include the same copy of the post-dominating code. The variable `pbra` is set, in the THEN and the ELSE paths, to either 0 (included) or 1 (excluded). It is checked by the thread that leaves the branch structure as to whether additional execution is required (lines 43 and 44).

Control-Flow Breaks

Control-flow breaks in the parent and child regions are handled in the following ways:

1. Conditional and unconditional branches or jumps. Trivial branches that are not speculated might be included in the parent or child regions during the pre-transformation analysis. If the branches are included in the child region, then their targets must also be inside the region. If they are included in the parent region but the targets are outside the region, then interrupt instructions (similar to line 19) are added. The child thread is aborted before the parent jumps to the outside targets.

2. Procedure returns and program exits. These breaks are typically guarded by conditional branches. They are only included in the parent region. Interrupt instructions are inserted before the breaks to abort the child thread. Then the parent exits the current procedure or stops the program execution.

3. Procedure calls. Only calls to non-recursive procedures are included. The procedure should not contain instructions that may raise exceptions. As dependence on data is not speculated, if the child consumes a value returned from a procedure which is called by the parent, it has to wait until that value is available. As *infinite-save-registers* option is used in the back-end code generator, the values passed to and from the procedures are saved in registers only. The procedure calls might be included in the child region although it is avoided.

4. Exceptions. The source code had been checked and modified to handle exceptions before it was translated into SUIF IR. Instructions that may cause exceptions are guarded by conditional branches. If a condition leading to an exception occurs inside a procedure, then a unique value is returned. At the caller's site, a conditional branch is also added to check whether the value is returned by an exception or a normal execution. In case that the current site is the main procedure, then the program execution stops. This is translated into SUIF IR as a series of procedure returns and program exit guarded by conditional branches.

5.1.2 Dual-Path Speculation

In contrast to the single-path speculation, the dual-path one does not predict the branch direction. Instead, threads are forked to speculatively execute both paths. The more probable one is forked earlier so that it can acquire any available TPU before the other.

The parent thread also executes the code in the parent region and the conditional code in parallel with the child threads' execution. As soon as the branch direction is known, one of them will proceed while the other will be squashed.

An example of code generated for dual-path speculation is shown in Figure 5.7. The transformer keeps two lists of dependent instruction pairs. Each of them is for data dependency between the parent and each child. Memory communication is handled in the same way as in the single-path speculation. Variables originally accessed by both paths are replicated to avoid the second speculative thread reading the value written by the first speculative thread. Otherwise, load operations in the second path are guarded by *safe/unsafe* switch indicating whether to load the safe version of data from the shared memory instead of searching through the speculative buffers (this applies to all threads, in the case of compound speculation).

If register communication is used, an `fregs` instruction broadcasts registers to both child threads at once. The registers forwarded from the parent will be received only if the *wait* bits in the child's registers are set to *TRUE*. Figure 5.8 is an example of register communication, in which both child threads are register dependent on different registers of the parent. The dependent registers are declared by a `uregs` instruction prior to each fork; however, the effect of `uregs` is cumulative. Thus, upon thread initialisation of the second child, both `$r1` and `$r2` are unavailable. During the computation, the *wait* bit in `$r1` is automatically set to *FALSE* upon the second thread's write-back operation. If `$r1` is forwarded from the parent before the write-back, then it is accepted but overwritten afterwards. Alternatively, `$r1` in the second path could be renamed.

```
1   int guard, T_fsucc, E_fsucc, pbra;                  // working variables
2   int myself, T_mychild, E_mychild, myparent;         // working variables
3   PAR_PREDICT:                                        //
4       guard = 1;                                      // inherited by child threads
5       T_fsucc = frk (sequence_no, THEN_LABEL);        // fork 1st speculative thread
6       T_mychild = cadr ();                            //
7       E_fsucc = frk (sequence_no, ELSE_LABEL);        // fork 2nd speculative thread
8       E_mychild = cadr ();                            //
9       guard = 0;                                      // indicate that this is parent thread
10      ... parent region's code ...                    //
11  PAR_VERIFY:                                         //
12      branch PAR_THEN                                 // original is ''branch THEN_LABEL''
13  PAR_ELSE:                                           //
14      isg (T_fsucc, T_mychild, CH_ROLLBACK_THEN);     // interrupt child's execution in THEN
15      psg (E_fsucc, E_mychild, sequence_no);          // pass signal to child in ELSE
16      sstp (E_fsucc, E_mychild);                      // parent synchronises and stops
17      goto ELSE_LABEL                                 //
18  PAR_THEN:                                           //
19      isg (E_fsucc, E_mychild, CH_ROLLBACK_ELSE);     // interrupt child's execution in ELSE
20      psg (T_fsucc, T_mychild, sequence_no);          // pass signal to child in THEN
21      sstp (T_fsucc, T_mychild);                      // parent synchronises and stops
22      goto THEN_LABEL                                 //
23  CH_PROLOGUE:                                        //
24  ELSE_LABEL:                                         //
25      pbra = 1;                                       // post-dominating instructions excluded
26      safe (guard, 0);                                // become speculative (unsafe)
27      ... ELSE path's code ...                        //
28      goto CH_RESOLVE;                                //
29  THEN_LABEL:                                         //
30      [ same as lines 25-26 ]                         // initialise thread
33      ... THEN path's code ...                        //
34  CH_RESOLVE:                                         //
35      if (!guard) goto DONE_LABEL;                    // if this is parent, exit
36      wat (guard, sequence_no);                       // if this is child, wait for the signal
37  CH_COMMIT:                                          //
38      cmmt (guard);                                   // commit speculative stores
39      goto DONE_LABEL;                                //
40  CH_ROLLBACK_ELSE:                                   //
41      ... abort slaves in ELSE ...                    //
42      stp (guard, -1);                                // child stops
43  CH_ROLLBACK_THEN:                                   //
44      ... abort slaves in THEN ...                    //
45      stp (guard, -1);                                // child stops
46  DONE_LABEL:                                         // exit
47  if (pbra)                                           //
48      { ... post-dominating region ... }              //
```

Figure 5.7: Code generated by **Spec_Transformer_2**

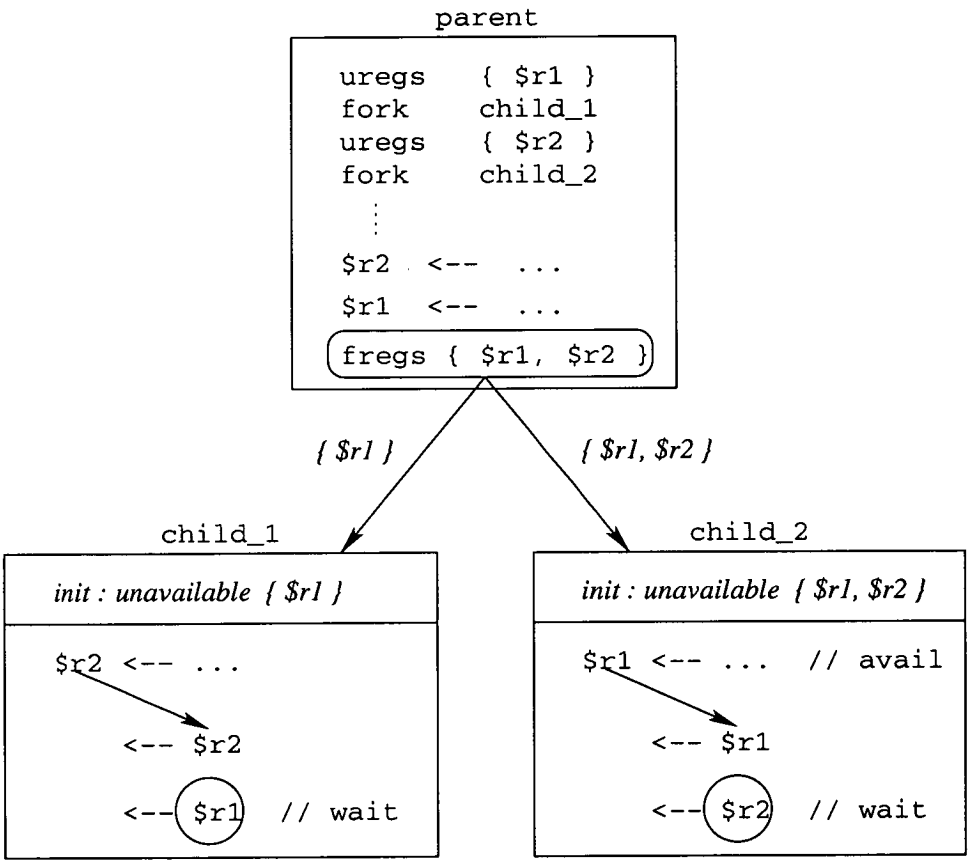


Figure 5.8: Register communication

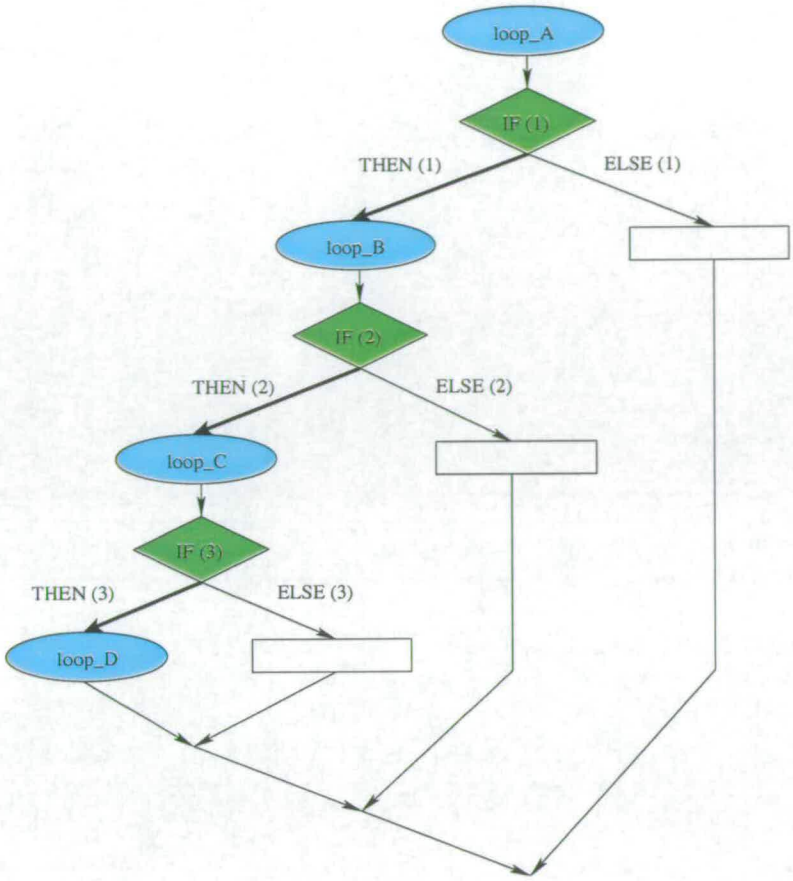


Figure 5.9: Sample nest of branches for Figure 5.10

5.1.3 Nested Speculation

The speculation is extended to nested branches. We focus on *completely-nested branch structures*. Interference from any other control-flow path was eliminated as a result of the code replication applied during the pre-transformation analysis.

Figure 5.9 gives an example of nested branches, in which paths *THEN(1)*, *THEN(2)*, and *THEN(3)* are predicted. The generated code (a skeleton is shown in Figure 5.10) is similar to the one produced by **Spec_Transformer_1** (or **Spec_Transformer_2**, in the case of dual-path speculation), but with a few additional constraints.

```
 1   ROOT_PROLOGUE:                          // executed by outermost branch
 2       out_guard = 0;                      //
 3       NEST_ID = sequence_no               // unique signal used in the nest

 4   int guard, fsucc, pbra;                 // local variables (per branch)
 5   int myself, mychild, myparent;          // local variables (per branch)
 6   PAR_PREDICT:                            //
 7           same as Figure 5.5              // fork a speculative thread
 8   PAR_VERIFY:                             //
*9       wat (out_guard, NEST_ID);           // wait until all outer branches resolved
10           same as Figure 5.5              // evaluate the branch
11   PAR_RIGHT:                              //
12       if (fsucc) {                        //
13           cmmt (1);                       //
*14          psg (fsucc, mychild, NEST_ID);  // pass signal to child
15           sstp (fsucc, mychild);          //
16       }                                   //
17       else goto THEN_LABEL                // ''goto ELSE_LABEL'' if ELSE is predicted
18   PAR_WRONG:                              //
19       isg (fsucc, mychild, CH_ROLLBACK);  // interrupt child's execution
*20      psg (1, myself, NEST_ID);           // deposit signal before proceeding
21       goto ELSE_LABEL;                    // ''goto THEN_LABEL'' if ELSE is predicted
22   CH_PROLOGUE:                            //
23       out_guard = 0;                      // switch off guard from outer branch
24           same as Figure 5.5              // initialise child thread
25   ELSE_LABEL:                             //
26       ... ELSE path's code ...            //
27       goto CH_RESOLVE;                    //
28   THEN_LABEL:                             // predicted path
29       ... THEN path's code ...            //
30   CH_RESOLVE:                             //
31       if (!guard) goto DONE_LABEL         //
32       wat (guard, NEST_ID);               // child waits for signal
33   CH_COMMIT:                              //
34       cmmt (guard);                       //
*35      psg (1, myself, NEST_ID);           // signal itself
36       goto DONE_LABEL                     //
37   CH_ROLLBACK:                            //
38       ... abort slaves in THEN ...        // or ELSE if it is predicted
*39      isg (in_fsucc, mychild, IN_CH_ROLLBACK); // interrupt next thread in the nest
40       stp (guard, -1);                    //
41   DONE_LABEL:                             //
```

Figure 5.10: Code generated for nested speculation

Firstly, the branches are resolved in sequential order. The parent thread is blocked (line 9) until it receives the signal from its own parent which speculates the previous or outer branch. Once the signal is received, it proceeds to evaluate the branch and pass the signal to its child if the speculation is correct (line 14) before stopping (line 15). Due to the default forking operation, the parent inherits all the guards from its predecessors and passes them to the child. The child leaves its own guard on but switches off the others (it only has to switch off the parent's guard). If the parent thread encounters any control-flow break from its parent region, it has to wait until the outer branches are resolved. Thus, a `wat` instruction similar to line 9 is inserted in front of the break.

Another constraint is how incorrect speculation is handled (line 18). A simple strategy has been implemented, i.e. the child thread and all its successors are aborted (line 39). The parent thread then executes the correct path (line 21). After the speculation is resolved, the thread that executes the correct path (either the parent or the child) will leave the current branch at `DONE_LABEL` (line 41) and arrive at `CH_RESOLVE` (line 30) of the outer branch.

Data Dependence

The handling of data dependence is slightly more complicated as the transformation of each branch is performed separately. An example is displayed in Figure 5.11(a). At run-time, the order of threads T_1, T_2, and T_3 is maintained by the global thread control unit (GTCU). T_3 should read A from T_1's buffer (since there is no store to A's address by T_2). Synchronisation between the grandparent (T_1) and grandchild (T_3) is required to ensure that the data retrieved by T_3 is the correct version.

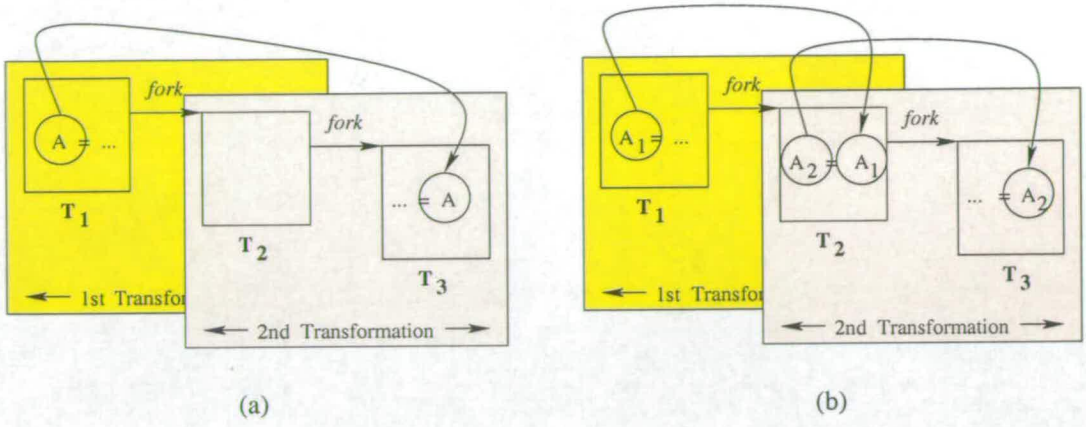


Figure 5.11: Handling of data dependencies in nested branches

We had opted to handle data dependencies and synchronisation on a parent/child basis. A copy instruction is therefore inserted in T_2's code, as shown in Figure 5.11(b), to convey the data from T_1 to T_3. The same strategy is applied if register communication is used in place of memory communication. Although simple to implement, a drawback of this method is the introduction of artificial data dependencies.

Table 5.2 Description and general statistics of synthetic benchmarks

Name	Description	Dynamic Instructions	Distribution (%)		
			Init	Main	Verify
SYN_1	Simple branch I	3,316,589	0.34	99.66	0.00
SYN_2	Simple branch II	3,847,163	0.37	99.63	0.00
SYN_3	A series of branches I	4,512,484	0.32	99.49	[a] 0.19
SYN_4	A series of branches II	4,516,761	0.37	99.63	0.00
SYN_5	A nest of branches I	3,846,046	0.34	99.56	0.10
SYN_6	A nest of branches II	4,481,544	0.35	99.65	0.00
SYN_7	Branch in multithreaded loop	4,682,408	0.12	99.88	0.00
	Average	4,171,856	0.32	99.64	0.05

[a] Sequential loop that computes `Lrt3` is included

5.2 Performance Evaluation

5.2.1 Benchmarks

Benchmarks used in the experiments were synthesised from modified Livermore loops which were arranged in conditional branch structures. Table 5.2 displays the general statistics of the benchmarks collected from their sequential execution. Figures 5.12 to 5.20 show the modified Livermore kernels (their average execution time are shown in Table 5.3) and fragments of the benchmarks' source code and control-flow graphs. The simulator takes as its input the assembly code of the benchmarks. It takes around 25-30 minutes to run a sequential program of 4.5 million dynamic instructions to completion, and up to 40-45 minutes for a multithreaded version of the same program (the overhead is due to the updating and searching of thread information in TPUs and GTCU).

Table 5.3 Average sequential execution time (per invocation)

Kernel	A_1	C_3	G_7	L_12
Time Units	56613	28056	176000	39500

```
// global variables
int xA[501], xC[501], xG[501], xL[501], y[501], z[523], u[523];
int r, t, q;
int Lrt1, Lrt2, Lrt3, kLrt1, kLrt2;
int L, LOOP, N, csum;

void init ()
{
    int k;
    for (k = 0; k <= 500; k++) {
        xA[k] = xG[k] = xL[k] = 0;
        xC[k] = 500 - k;
        y[k] = 1;
    }
    for (k = 0; k <= 522; k++) {
        u[k] = k;
        z[k] = k + k;
    }
    r = 5; t = 2; q = 0;
}

void A_1 (int n1, int n2)                                    // Loop A
{
    for (int k = n1; k < n2; k++) {
        xA[k] = q + y[k] * (r * z[k + 10] + t * z[k + 11]);
    }
}

int C_3 (int n1, int n2, int mc1, int mc2)                   // Loop C
{
    rC = 0;
    for (int k = n1; k < n2; k++) {
        rC = rC + z[k] * xC[k];
        rC = rC + mc1 + mc2;
    }
    return rC;
}
```

Figure 5.12: Modified Livermore kernels (continued in Figure 5.13)

```
void G_7 (int n1, int n2, int mg)                                    // Loop G
{
    for (int k = n1; k < n2; k++) {
        xG[k] = u[k] + r * (z[k] + r * y[k]) +
                t * (u[k + 3] + r * (u[k + 2] + r * u[k + 1]) +
                t * (u[k + 6] + r * (u[k + 5] + r * u[k + 4])));
        xG[k] = xG[k] - mg;
    }
}
void L_12 (int n1, int n2, int ml1, int ml2)                         // Loop L
{
    for (int k = n1; k < n2; k++) {
        xL[k] = (ml1 * y[k + 1]) - (ml2 * y[k]);
    }
}
```

Figure 5.13: Modified Livermore kernels (continued from Figure 5.12)

Four Livermore loops were used: *L*, *A*, and *G* are small, medium, and large fully-parallelisable loops, respectively; *C* is a small loop with cross-iteration dependence. The structures of the synthetic benchmarks provide opportunities for single-path, dual-path, and nested speculation. The first six benchmarks can be divided into two groups: $\{SYN\_1, SYN\_3, SYN\_5\}$ and $\{SYN\_2, SYN\_4, SYN\_6\}$. The second group imitates the first one, but it provides further opportunities to speculate on control-independent paths of the branches (Section 5.2.2.2). Branch probabilities and loop sizes in the parent and the child regions are varied between the benchmarks, in order to explore different resource allocation strategies.

- *SYN_1* and *SYN_2* contain simple branch structures. In *SYN_1*, the loops inside the branch structure (speculative) are bigger than the one dominating the branch (non-speculative), while the opposite is the case in *SYN_2*.

```
main ()
{
LOOP = 70; N = 501; csum = 0;
for (L = 1; L <= LOOP; L++) {
    Lrt1 = (L + r) * t;
    Lrt2 = (L * r) + t;
    A_1 (0, N);
    if ((xA[N-1] * Lrt1) > (xA[N/2] * Lrt2))
        G_7 (1, N, Lrt1);
    else
        G_7 (1, N, Lrt2);
    for (k = 1; k < N; k = k+50)
    csum = csum + xA[k] + xG[k];
}
}
```

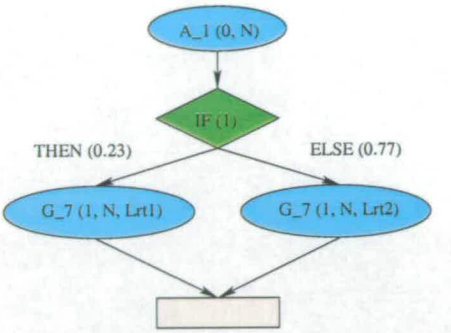


Figure 5.14: Synthetic benchmark *SYN_1*

- *SYN_3* and *SYN_4* contain a series of branch structures. They are used for testing if the code generated from the transformers work correctly, i.e. the structures must be handled one-by-one. Hence, the first structures and their threads must be resolved before subsequent ones are speculated.

- *SYN_5* and *SYN_6* contain nests of branch structures. The loops inside the inner branch structures (speculative) are the biggest ones in *SYN_5*, but the smallest ones in *SYN_6*.

The last benchmark *SYN_7* contains a parallisable loop, inside which is a branch structure. At present, the loop transformers and the speculation transformers work separately, and control dependence across loop iterations are neither recognised nor handled by the loop transformers. Thus, an assumption being made is that the branches inside the iterations must be independent of each other.

```
main ()
{
LOOP = 70; N = 501;
for (L = 1; L <= LOOP; L++) {
    Lrt1 = (L + r) * t;
    Lrt2 = (L * r) + t;
    A_1 (0, N);
    if ((xA[N-1]*Lrt1) > (xA[N/2]*Lrt2))
        L_12 (0, N-1, Lrt1, Lrt2);
    else
        L_12 (0, N-1, Lrt2, Lrt1);
    G_7 (1, N, 0);
    for (k = 1; k < N; k = k+50)
        csum = csum + xA[k]+xG[k]+xL[k];
}
}
```

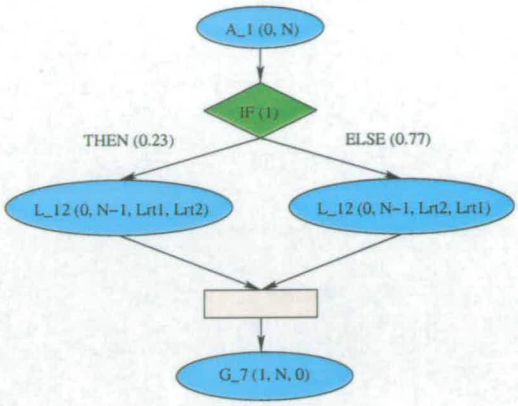


Figure 5.15: Synthetic benchmark *SYN_2*

```
main ()
{
LOOP = 70; N = 501;
for (L = 1; L <= LOOP; L++) {
    Lrt1 = (L + r) * t;
    Lrt2 = (L * r) + t;
    A_1 (0, N);
    if ((xA[N-1] * Lrt1) > (xA[N/2] * Lrt2))
        G_7 (1, N, Lrt1);
    else
        G_7 (1, N, Lrt2);
    Lrt3 = 0;
    for (k = 1; k < N; k++)
        Lrt3 = Lrt3 + 3 * xG[k] / (xA[k] + 1);
    if ( Lrt3 > (Lrt1 * Lrt2) )
        L_12 (0, N-1, r, t);
    else
        L_12 (0, N-1, t, r);
    for (k = 1; k < N; k = k+50)
        csum = csum + xA[k] + xG[k] + xL[k];
}
}
```

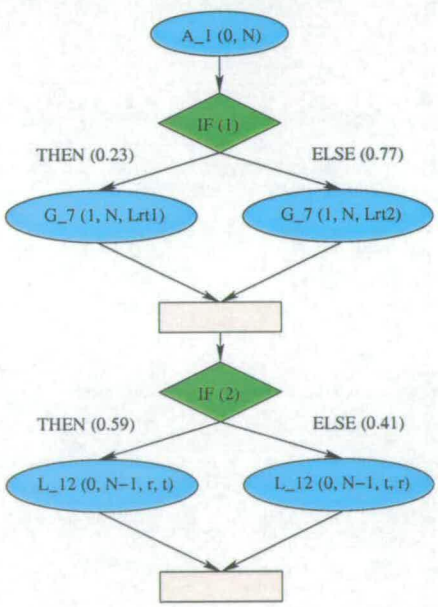


Figure 5.16: Synthetic benchmark *SYN_3*

```
main ()
{
LOOP = 70; N = 501;
for (L = 1; L <= LOOP; L++) {
    Lrt1 = (L + r) * t;
    Lrt2 = (L * r) + t;
    A_1 (0, N);
    if ((xA[N-1] * Lrt1) > (xA[N/2] * Lrt2))
        G_7 (1, N, Lrt1);
    else
        G_7 (1, N, Lrt2);
    qC = C_3 (0, N, -Lrt1, -Lrt2);
    if ( (qC/500) > (Lrt1*Lrt1 + Lrt2*Lrt2) )
        L_12 (0, N-1, r, t);
    else
        L_12 (0, N-1, t, r);
    for (k = 1; k < N; k = k+50)
        csum = csum + xA[k] + xG[k] + xL[k];
}
}
```

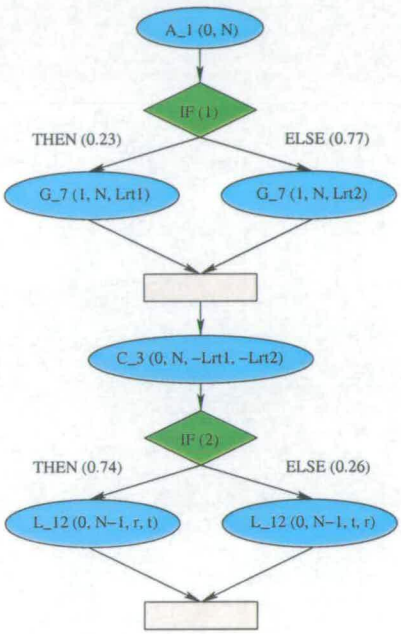


Figure 5.17: Synthetic benchmark *SYN_4*

```
main ()
{
LOOP = 70; N = 501;
for (L = 1; L <= LOOP; L++) {
    Lrt1 = (L + r) * t;
    Lrt2 = (L * r) + t;
    A_1 (0, N);
    if ((xA[N-1] * Lrt1) > (xA[N/2] * Lrt2)) {
        qC = C_3 (0, N, Lrt1, -Lrt2);
        if ( (qC/800) > (Lrt1 * Lrt1 * Lrt2) )
            G_7 (1, N, Lrt1);
        else
            G_7 (1, N, -Lrt1);
    }
```

... continued

```
    else {
        qC = C_3 (0, N, -Lrt1, Lrt2);
        if ( (qC/800) < (Lrt2 * Lrt2) )
            G_7 (1, N, -Lrt2);
        else
            G_7 (1, N, Lrt2);
    }
    for (k = 1; k < N; k = k+50)
        csum = csum + xA[k] + xG[k];
}
}
```

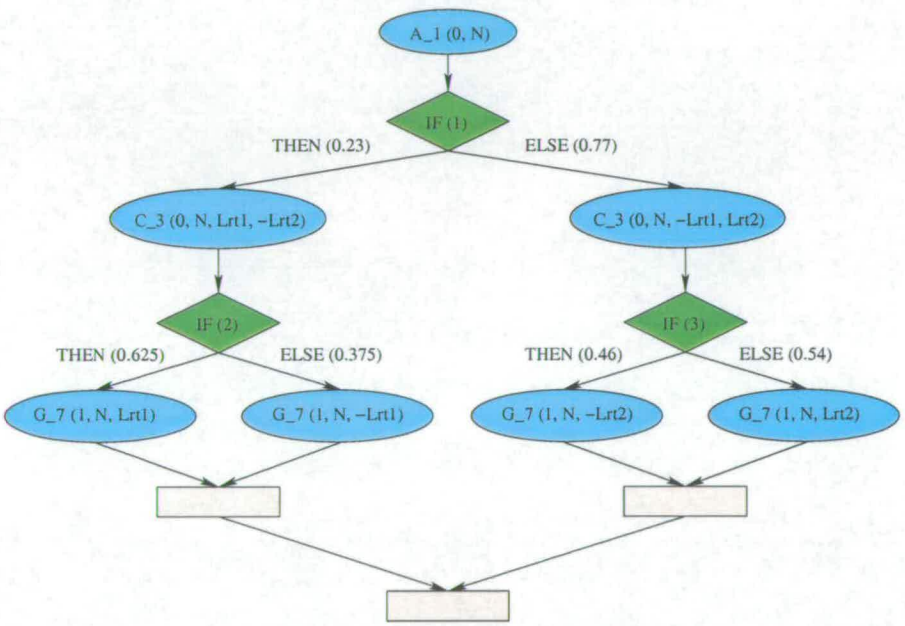


Figure 5.18: Synthetic benchmark *SYN_5*

```
main ()
{
LOOP = 70; N = 501;
for (L = 1; L <= LOOP; L++) {
    Lrt1 = (L + r) * t;
    Lrt2 = (L * r) + t;
    A_1 (0, N);
    if ((xA[N-1]*Lrt1) < (xA[N/2]*Lrt2)) {
        qC = C_3 (0, N, Lrt1, 0);
        if ( (qC % Lrt1) > (Lrt1 / r) )
            L_12 (0, N-1, r, 2*t);
        else
            L_12 (0, N-1, t, 2*r);
        G_7 (1, N, Lrt1);
    }
```

... continued

```
    else {
        qC = C_3 (0, N, Lrt2, 0);
        if ( (qC % Lrt2) > (Lrt2 / r) )
            L_12 (0, N-1, 2*r, t);
        else
            L_12 (0, N-1, 2*t, r);
        G_7 (1, N, Lrt2);
    }
    for (k = 1; k < N; k = k+50)
        csum = csum + xA[k]+xG[k]+xL[k];
}
}
```

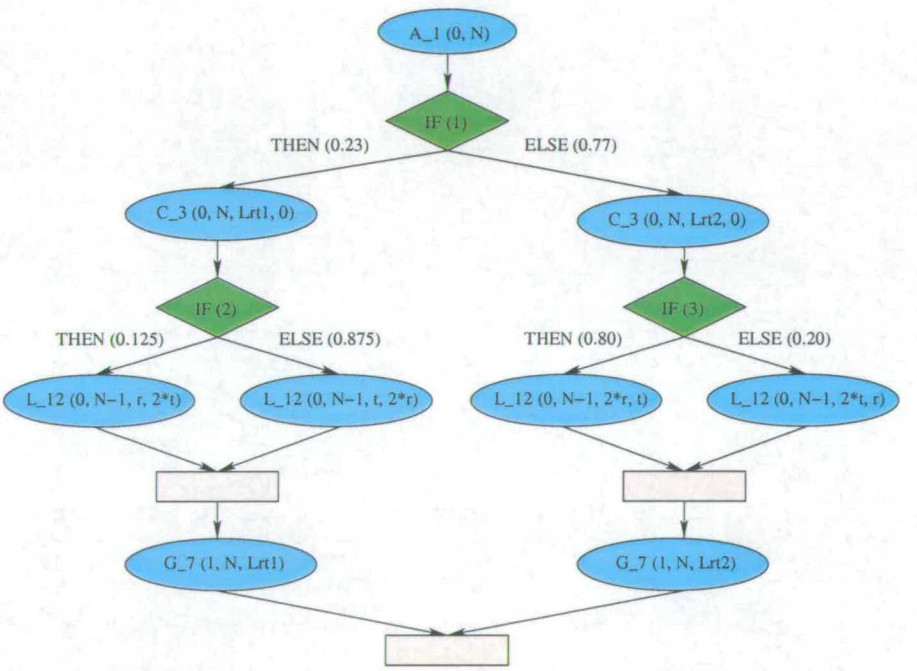


Figure 5.19: Synthetic benchmark *SYN_6*

```
main ()
{
    LOOP = 30; N = 51;
    for (L = 1; L <= LOOP; L++) {
        for (k = 0; k < N; k++) {
            xA[k] = q + y[k] * (r * z[k + 10] + t * z[k + 11]);  // A_1
            kLrt1 = k * (L + r + t);
            kLrt2 = k + (L * r * t);
            if (xA[k] > (kLrt1 + kLrt2))
                qC = C_3 (0, 201, kLrt1, -kLrt2);
            else
                qC = C_3 (0, 201, -kLrt1, kLrt2);
            xA[k] = xA[k] + qC;
        }
        for (k = 1; k < N; k = k+10)
            csum = csum + xA[k];
    }
}
```

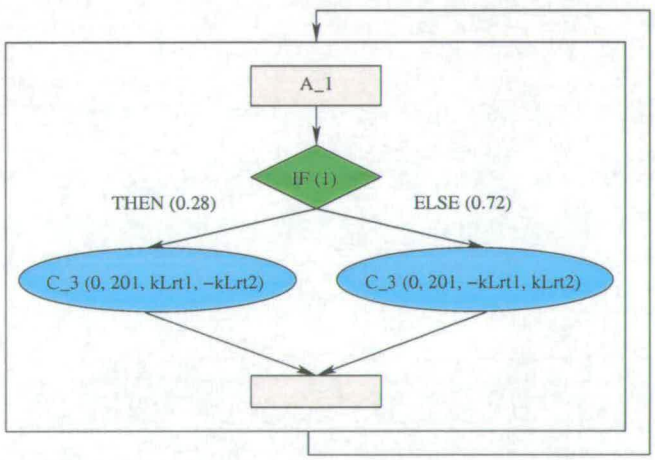


Figure 5.20: Synthetic benchmark *SYN_7*

5.2.2 Results and Discussions

Calls to the Livermore procedures were inlined so that the main procedures (`main()`) contain the complete Livermore loops. The loops in all the benchmarks, with the exception of *SYN_7*, are unrolled 99 times and re-rolled to produce chunks of 100 iterations each. For loops *A* and *C*, the first chunks in each case contain 101 iterations. When the loop is multithreaded, the first chunk is allocated to the master thread while the others are distributed to the slaves. The architectural parameters used in the simulation are the same as those listed in Table 4.3, except that the total number of TPUs is increased to 24. The probability threshold is set to 0.65, which implies that if the more probable path of a branch is less confident than this threshold, then the branch will be transformed for dual-path speculation. Since the synthetic benchmarks are well-structured, the pre-transformation processing which involves control-flow analysis, region formation, and dependency analysis is straightforward.

The performance of multithreaded non-speculative and speculative programs were compared. During the execution of control-independent loops in both cases, the slave TPUs are reusable. For the speculative execution of control-dependent loops, the reusability of the slave TPUs depends on when the (speculative) master threads receive synchronisation signals from their parents. A loop is *control-independent* of a branch if it dominates or post-dominates that branch, and is *control-dependent*, otherwise.

The first set of results is displayed in Figure 5.21. In both non-speculative (*MULTI*) and speculative (*SPEC*) programs, the sizes of clusters executing the control- independent loops range from 3 to 6 TPUs, whereas the sizes of those executing the control-dependent loops are fixed at 3 TPUs. This allocation strategy (see Section 5.2.2.1) is called *CIndep* as more TPUs are given to the control-independent partitions.

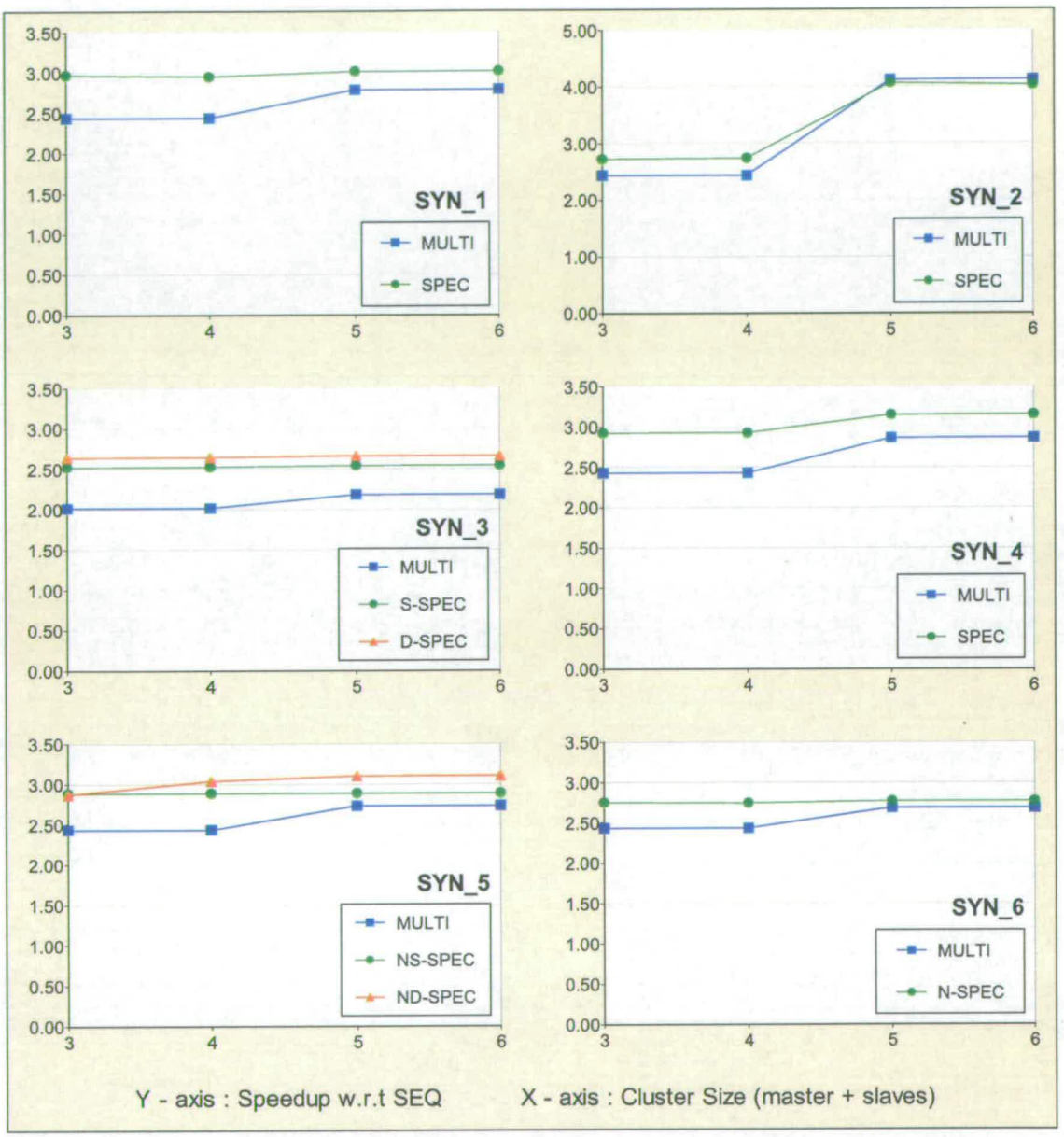


Figure 5.21: Speedup of speculative programs (*CIndep* policy)

For the speculative programs, prefix "*N-*" refers to nested speculation, "*S-*" single-path speculation in spite of branches' low confidence, and "*D-*" dual-path speculation in cases of branches' low confidence. The speculative execution offers slight improvement over the non-speculative one. Single-path speculation in spite of the branches' low probability causes frequent misprediction. Its penalty is in the lost opportunities of executing the correct paths in parallel with the parent threads' execution. From the graphs, it seems that the opportunities lost have little impact since the performance of *S-SPEC* and *NS-SPEC* in *SYN_3* and *SYN_5* is marginally poorer than *D-SPEC* and *ND-SPEC* in the same benchmarks.

5.2.2.1 Cluster Allocation

The performance of the multithreaded non-speculative programs shown earlier is below its true potential, i.e. the maximum speedup it could have achieved, given the total number of the TPUs available. The cluster allocation in these programs corresponds to the scheme used in their speculative counterparts. However, it is unfair when the nature of the non-speculative execution is considered, i.e. the loops are executed one-by-one. Because the control-dependent loops are executed after the branch directions are known, they can in fact reuse all the TPUs released by the control-independent loops. Figure 5.22 shows speedup of the non-speculative programs when all the loops are allocated the same number of TPUs ranging from 3 to 6. The increase in the speedup is significant when the number of TPUs matches the number of threads executing the loops.

A similar scheme can be used in the speculative programs. The difference is that only the control-dependent loops on the non-speculative paths can reuse all the TPUs because they are executed after misprediction occurs and they are confirmed to be the

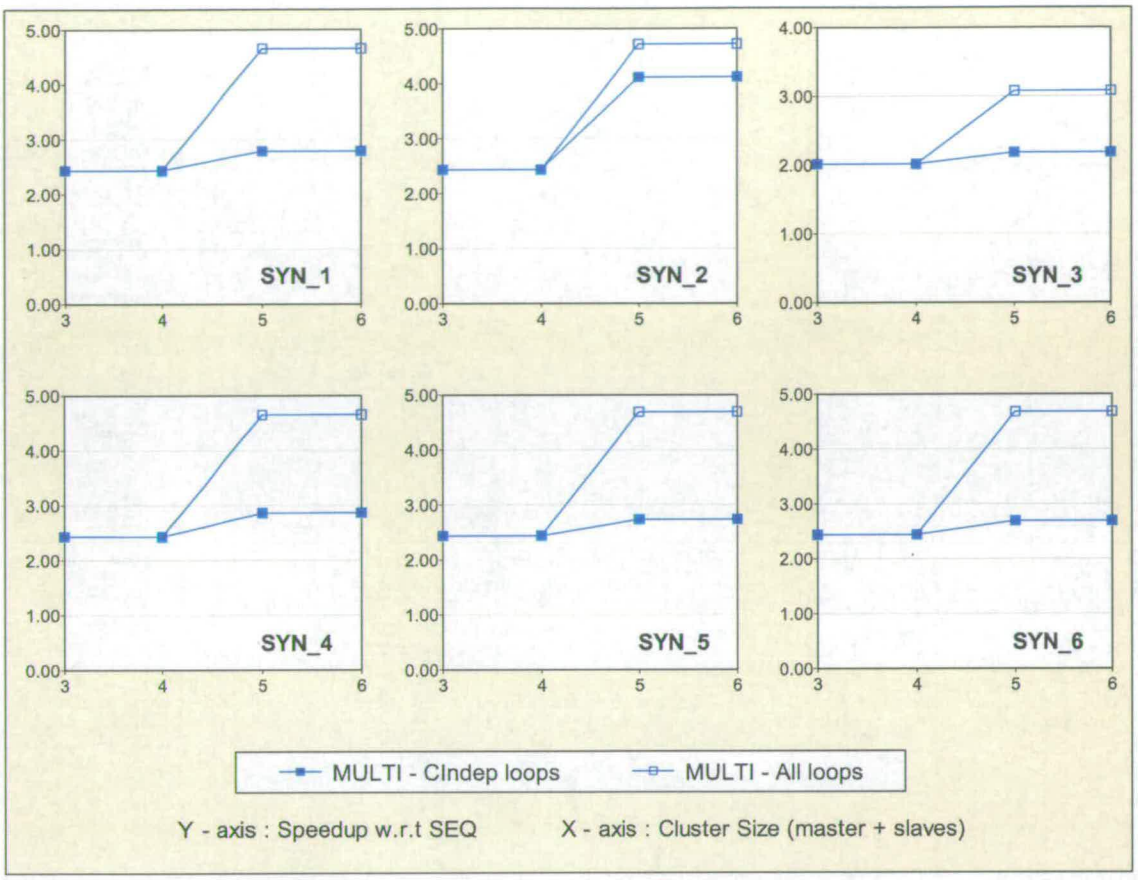


Figure 5.22: A comparison of 2 cluster allocation policies for non-speculative programs

correct paths. Ideally, these loops should reuse the TPUs released from both control-independent loops and the loops on the mispredicted paths. However, unless synchronisation is added, the loops on the correct paths may try to form clusters before the ones on the wrong paths release theirs. If the cluster sizes are larger than the number of TPUs guaranteed to be available when the misprediction recovery starts, then these operations may be unsuccessful, causing the loops to be sequentially executed instead.

Another cluster allocation strategy considers the contribution of each loop to the overall program execution. If multiple loops are executed concurrently, the one that contributes most to the total execution time should receive the largest number of TPUs. To calculate the amount of each loop's contribution, the cumulative probability along the control-flow path leading to that loop and its execution time are taken into account. For each loop i,

$$T_i = \text{cumulative probability} \times \text{sequential execution time}$$

$$\text{contribution (\%)} = \frac{T_i}{\sum T_i} \times 100$$

Four cluster allocation strategies are examined. They employ different criteria to prioritise loops in the benchmarks. The highly-prioritised ones are given numbers ranging from 4 to 6 TPUs[3], whereas the others are always given only 3 TPUs. These strategies are

- *CIndep*. The priority is given to the control-independent loops only.
- *NonSPEC*. The priority is given to the non-speculative loops. If multiple loops are executed at the same time, the prioritised one is usually control-independent. The other non-speculative loops executed individually are also prioritised.
- *Critical*. If multiple loops are executed at the same time, then the priority is given to the loop which contributes most to the overall program execution. The contribution factor of each loop in the benchmarks are calculated and shown in Table 5.4.
- *All*. The same number of TPUs (3-6) is allocated to all the loops.

[3]All prioritised loops in a benchmark are given the same amount of TPUs, for example, all of them are given 6 TPUs.

Table 5.4 Contribution of individual loop to the overall program execution

Benchmark	Loop { *path* }	%	Loop { *path* }	%
SYN_1	A {*control indep*}	24		
	G {*THEN*}	17	G {*ELSE*}	58
SYN_2	A {*control indep*}	21	L {*THEN*}	3
	G {*control indep*}	65	L {*ELSE*}	11
SYN_3	A {*control indep*}	21		
	G {*THEN*}	15	L {*THEN*}	9
	G {*ELSE*}	50	L {*ELSE*}	6
SYN_4	A {*control indep*}	19	C {*control indep*}	9
	G {*THEN*}	13	L {*THEN*}	10
	G {*ELSE*}	45	L {*ELSE*}	3
SYN_5	A {*control indep*}	22		
	C {*THEN*}	2	C {*ELSE*}	8
	G {*THEN, THEN*}	10	G {*ELSE, THEN*}	24
	G {*THEN, ELSE*}	6	G {*ELSE, ELSE*}	28
SYN_6	A {*control indep*}	19		
	C {*THEN*}	2	C {*ELSE*}	7
	L {*THEN, THEN*}	<1	L {*ELSE, THEN*}	8
	L {*THEN, ELSE*}	3	L {*ELSE, ELSE*}	2
	G {*THEN*}	14	G {*ELSE*}	45

The results are shown in Figures 5.23 and 5.24. Generally, each speedup bar in the graphs has 4 layers. The bottom layer is the minimum speedup achieved by one of the 4 strategies, and the other 3 layers are the successive improvements of the other strategies over the previous ones.

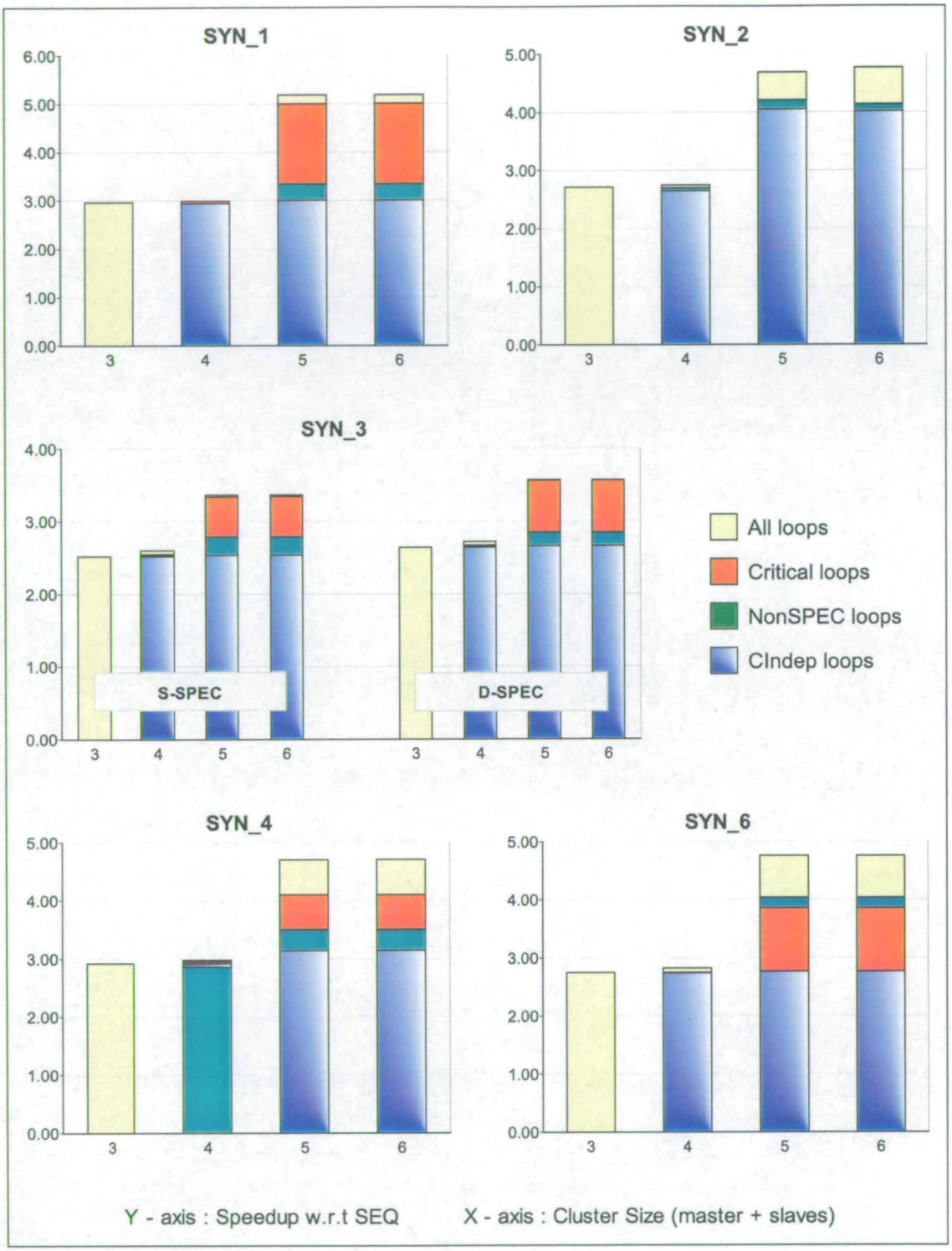


Figure 5.23: A comparison of 4 cluster allocation policies for speculative programs

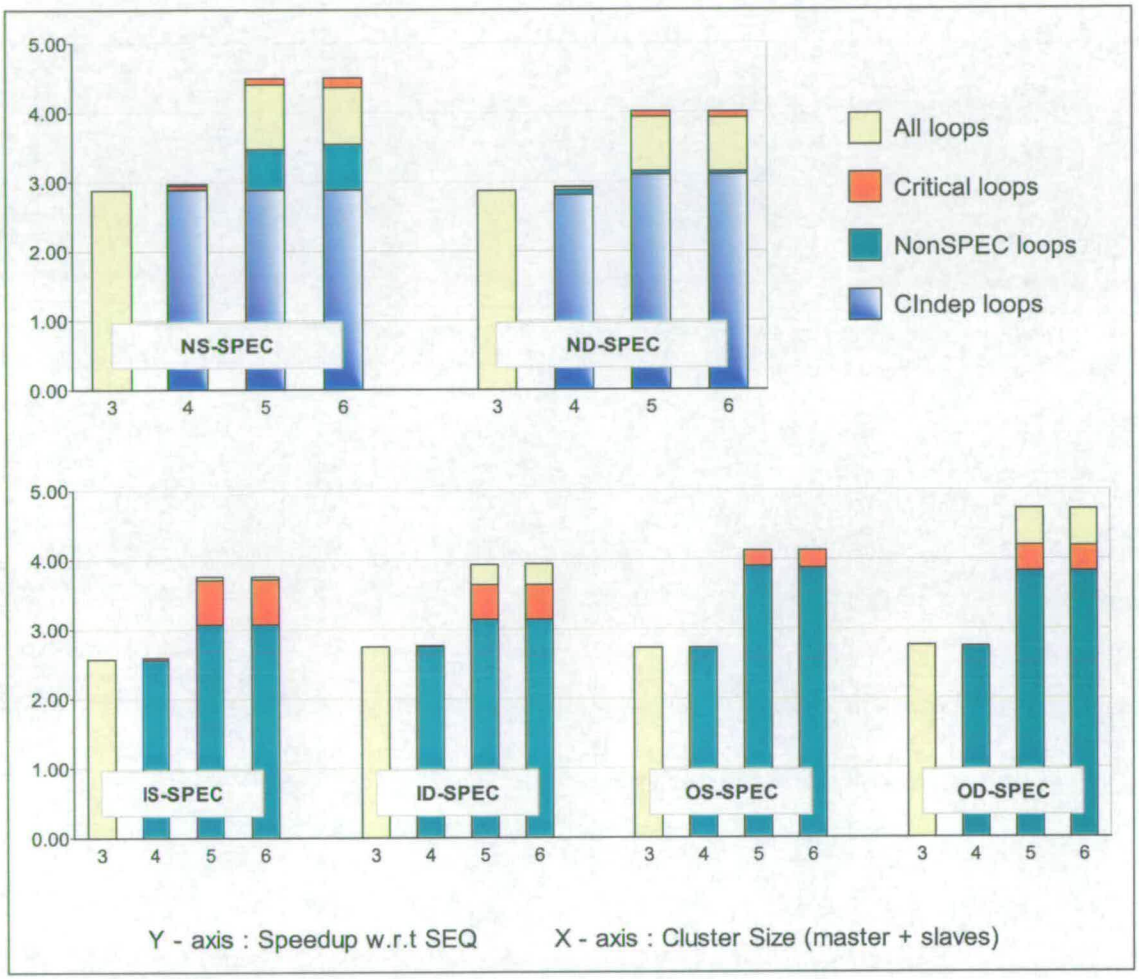


Figure 5.24: A comparison of 4 cluster allocation policies for the nested speculation in *SYN_5*

From Figure 5.23, the *All* strategy performs best, followed by *Critical*, *NonSPEC*, and *CIndep*. This is clearly seen when the prioritised loops are given at least 5 TPUs, which allows all the threads to be successfully sparked. In *SYN_6*, *NonSPEC* performs a little better than *Critical*. Both strategies prioritise loops *A* and *G* which are the main contributors to the total execution time. However, they have different views when choosing between loops *C* and *L*, i.e. *NonSPEC* favours *C* while *Critical* favours *L*. If both loops in question have little impact on the total execution time, it seems that the non-speculative one should be favoured because its results are at least guaranteed to be useful. For all the benchmarks, in general, the biggest improvement step comes from the *Critical* strategy. In *SYN_2*, the cluster allocation by *CIndep* is identical to the one by *Critical*; therefore, it appears in this figure (and Figure 5.21) that *CIndep* already gives good speedup.

The speedup of nested speculation in *SYN_5* is shown in the first graph of Figure 5.24. When there are loops from multiple nest levels being executed at the same time, dual-path speculation holds back the performance improvement as it allows even more loops to compete for available TPUs. Although the total number of TPUs (24) seems to be sufficient, during the run-time, some cluster or fork operations are executed a little too early or too late in relation to the availability of resources. This happens especially when there are several multithreadable loops active simultaneously. The loop on the secondary path is often the last one attempting to form a cluster and thus it is most likely to fail. If the secondary path is correct, then the benefit from it having been partially executed is outweighed by the remaining execution being sequential. On the other hand, single-path speculation delays the less probable path until the misprediction recovery takes place, but it can gain more from the multithreaded execution on this path. It appears that in both single- and dual-path speculation, the *Critical* strategy

performs slightly better than *All* as a result of fewer simultaneously-executed loops competing for the TPUs.

In the second graph of Figure 5.24, either the inner or the outer branch in the nest is speculated. *IF(2)* is always speculated because it is not handled in parallel with *IF(1)* and *IF(3)*. The prefix "*I-*" or "*O-*" indicates whether the inner or the outer branch is chosen. Having learnt that the performance of *CIndep* is only, at best, as good as *NonSPEC*'s, it is excluded from the experiment. When there are sufficient TPUs to execute several loops simulteneously, dual-path speculation yields higher speedup than single-path speculation. Furthermore, outer-branch speculation yields higher speedup than inner-branch speculation. This can be explained by the fact that the branch in the deeper nest level is less likely to be encountered and is therefore less profitable to speculate.

5.2.2.2 Control-Independent Execution

In addition to speculating on control-dependent paths of a branch, another thread can be launched to execute the code after those paths converge. Although the code is to be executed regardless of the branch's direction, the thread as well as its children and slaves are speculative because this program fragment may be on either path of another branch. *SYN_2*, *SYN_4*, and *SYN_6* are used for studying the impact of control-independent execution. In *SYN_6*, loop *G* is control-dependent on the outer branch but independent of the inner branches. The transformation is adapted from the one that generates single-path speculative programs.

Figure 5.25 illustrates the four major sections in the control-independent (CI) and the control-dependent, speculative (CSP) execution. These relate to the points where a new thread is forked (`PAR_PREDICT`) and initialised (`CH_PROLOGUE`), and where the flow

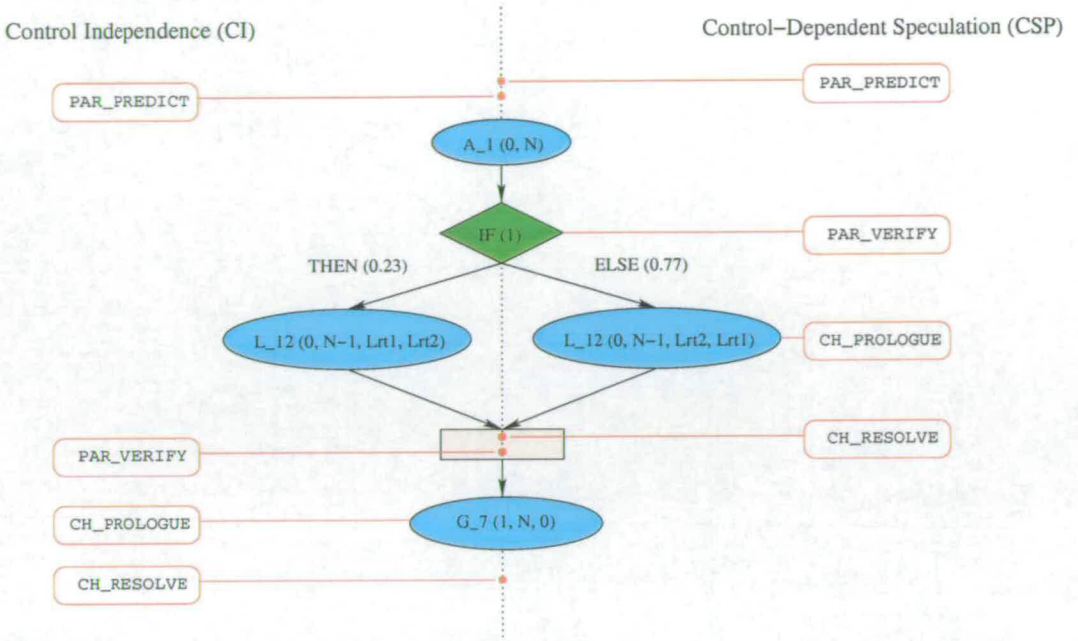


Figure 5.25: An outline of control-independent execution in *SYN_2*

of control is transferred from parent to child threads (`PAR_VERIFY` and `CH_RESOLVE`). The order in which CI and CSP threads are forked may also affect the program performance since the first thread can compete for the TPUs before the other.

In Figure 5.26, two cluster allocation strategies, *NonSPEC* and *Critical*, are employed. *CI-CSP* and *CSP-CI* indicate the order in which the CI and CSP threads are forked. This order is not significant when a reasonably large amount of TPUs are present. Comparing the best results from the *Critical* scheme to the best results from Figure 5.23, where only the CSP is performed, it appears that the CI technique further boosts the program speedup.

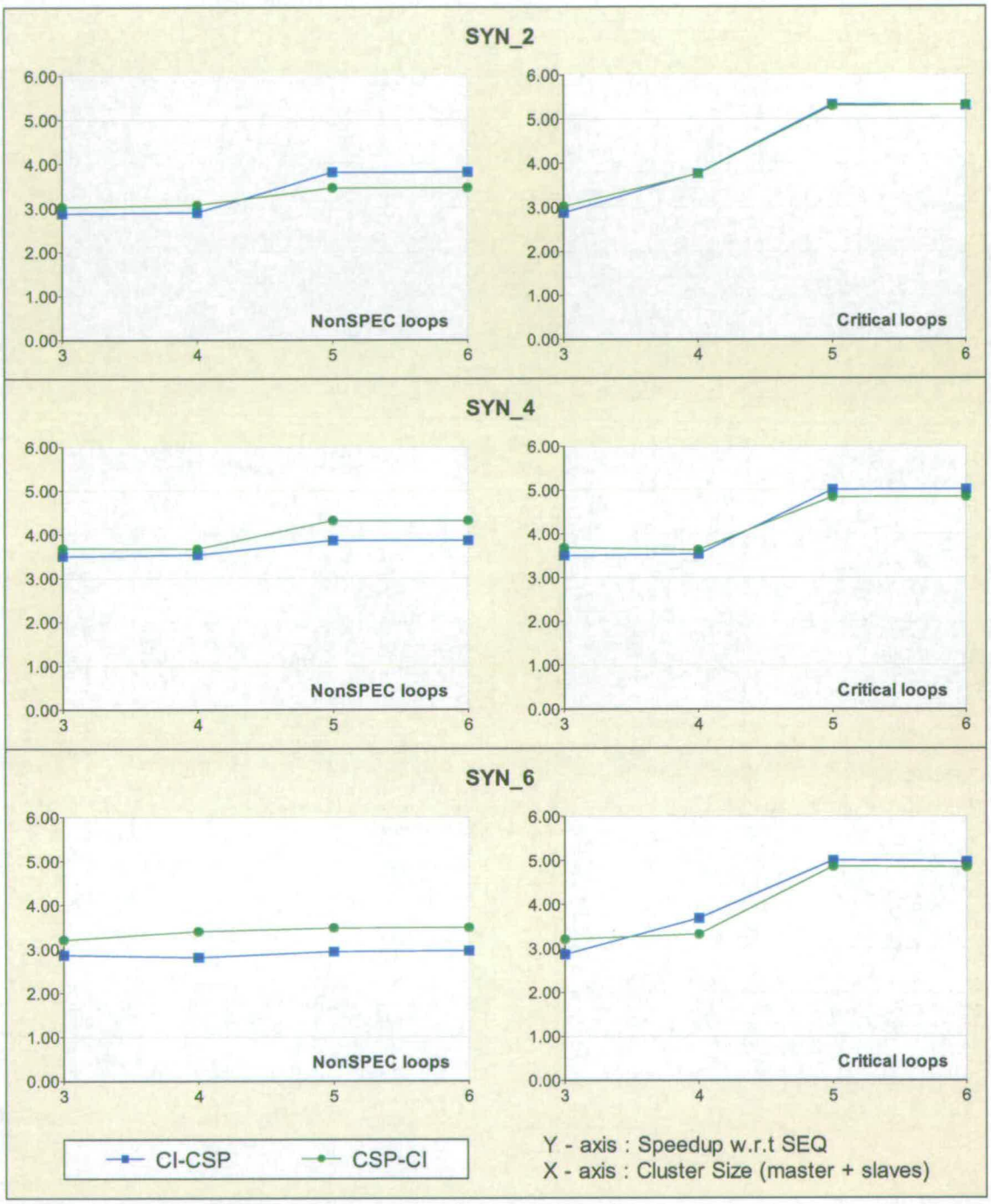


Figure 5.26: Speedup after CSP and CI are performed (total TPUs = 24)

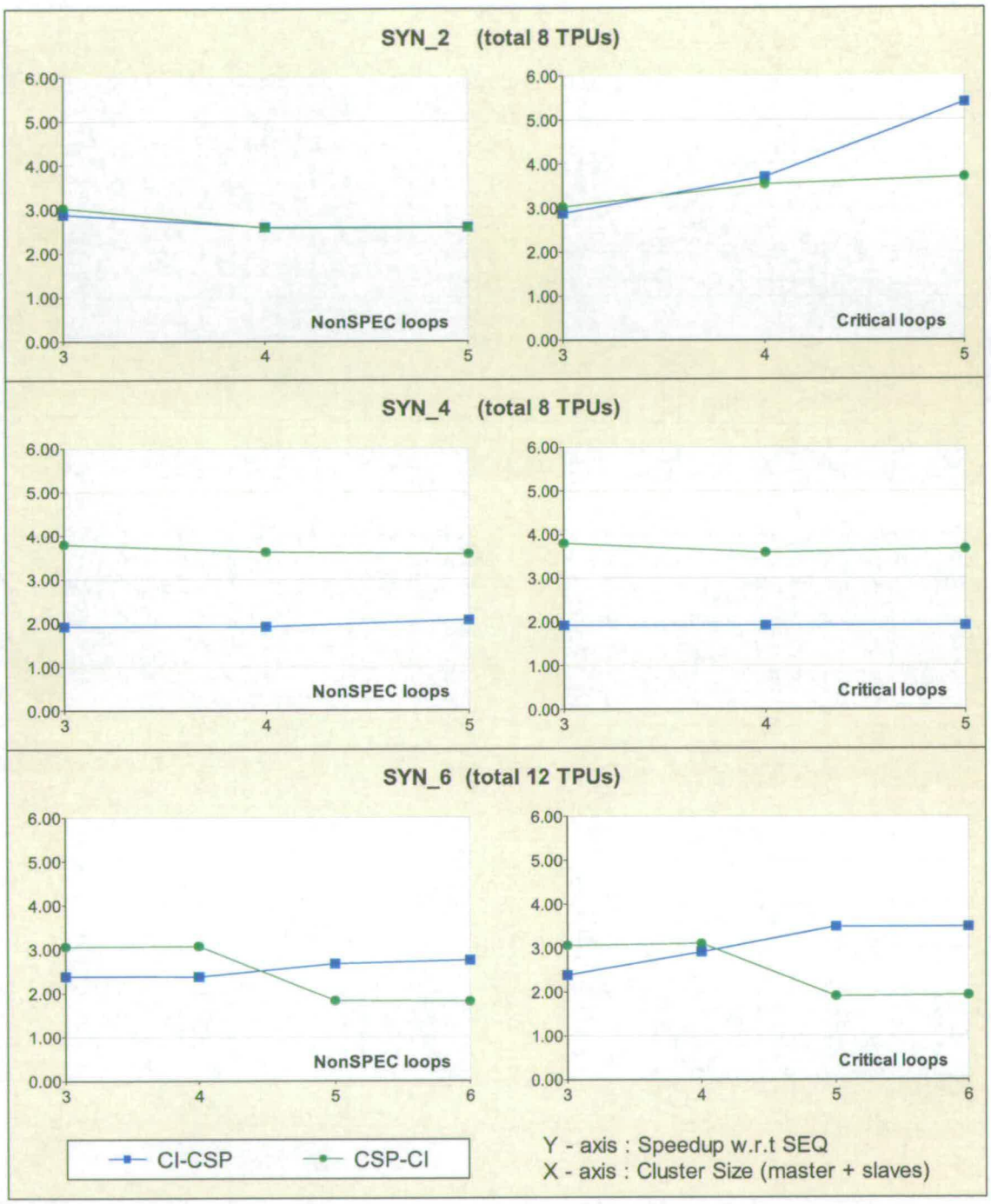


Figure 5.27: Speedup after CSP and CI are performed (total TPUs = 8, 12)

The order of forking between CI and CSP threads has a visible impact when the total number of TPUs decreases, as displayed in Figure 5.27. In *SYN_2* and *SYN_6*, the CI threads execute dominant loops whereas the CSP threads execute trivial loops. Thus, it is more beneficial to allow the CI threads to acquire the TPUs first. In *SYN_4*, it is the opposite case. The CI threads not only execute the trivial loops themselves, but also fork threads to speculatively execute other trivial loops which are further ahead. As a result, the CSP threads which execute the most dominant loops are hindered when the *CI-CSP* policy is used.

Knowing that loops on the control-dependent paths of the branch in *SYN_2* and the inner branches in *SYN_6* are the least dominant in the programs, we tested only the CI technique but omitted the CSP one. The *Critical* strategy determines which loops among those simultaneously executed should receive more TPUs. The results were plotted against the best ones from *CSP* (due to *All* strategy in Figure 5.23) and *CSP+CI* (due to *Critical* strategy in Figure 5.26), and shown in Figure 5.28. In both the benchmarks, the highest speedups were achieved by performing only control-independent execution.

In spite of having a similar program structure to the one in *SYN_4*, the CI region in *SYN_3* (which dominates the first branch and post-dominates the second one) consumes results from both control-dependent paths of the first branch. As data speculation is not supported, this region can only be executed after the first branch is resolved, but the lookahead speculation can be performed by speculating the second branch (or launching its CSP threads) immediately after the first one. However, the results in Figure 5.29 when compared with those in Figure 5.23 show that earlier execution of the lookahead paths, in this case, yields no further improvement since the loops on these paths are very small.

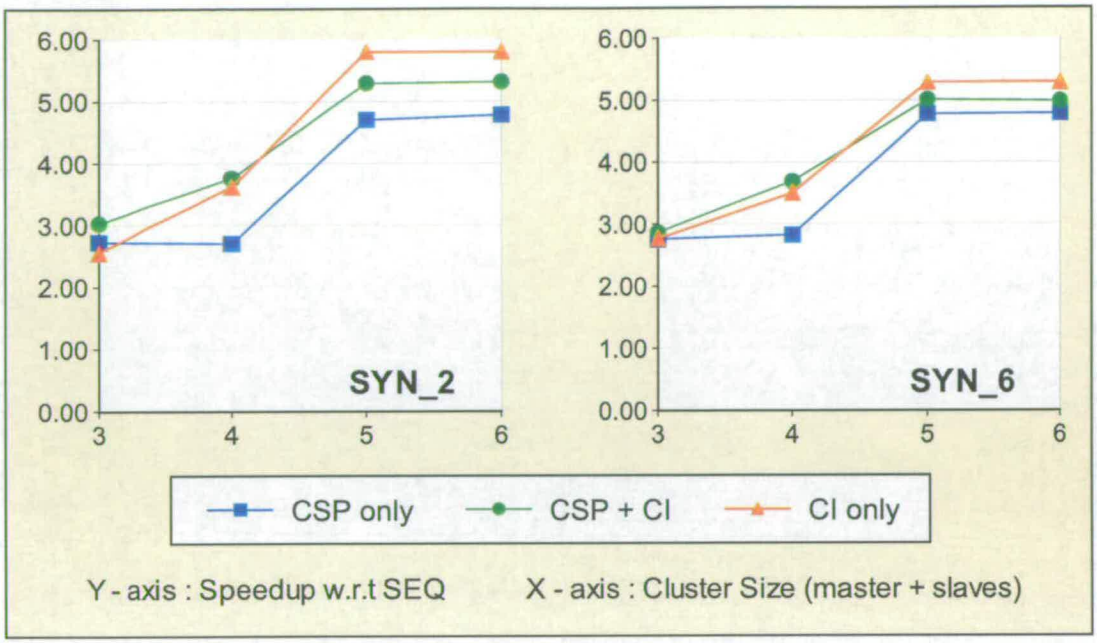


Figure 5.28: Best performance from CSP, CI, and CSP+CI

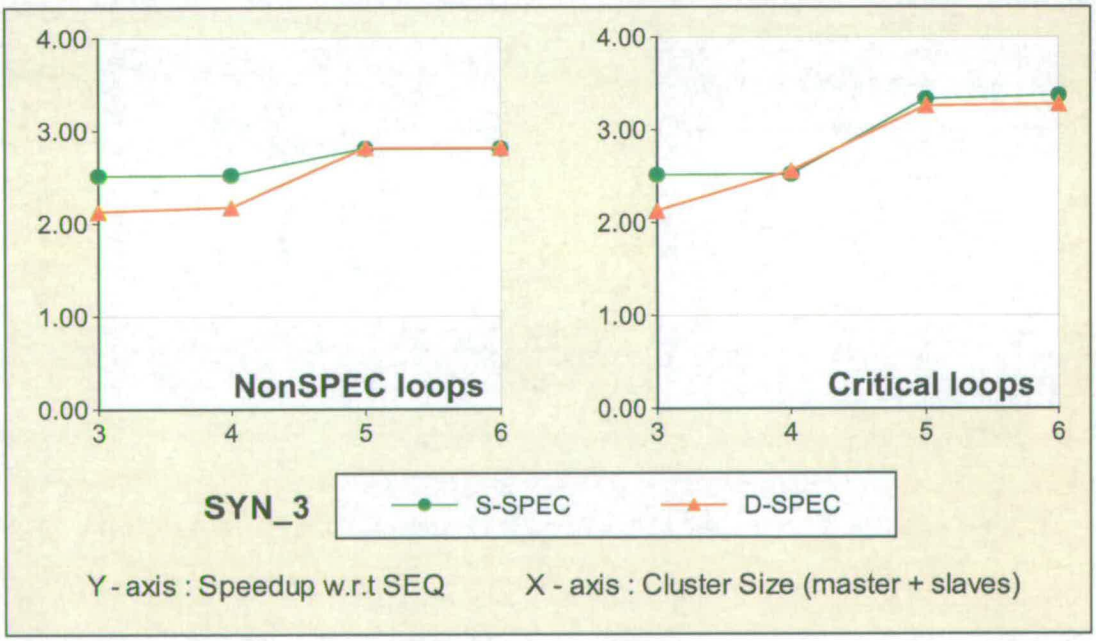


Figure 5.29: Results from the lookahead speculation

5.2.2.3 Concurrent Speculation

Unique among the benchmarks, the branch in *SYN_7* resides in the body of the outer loop which is multithreadable. Several instances of the branch can be speculated at the same time as they are independent of each other. Additionally, neither the branch nor the execution of its control-dependent paths causes premature exit from the outer loop. In Figure 5.30, *N-MULTI* allows multithreading in both the outer and the inner loops; *O-MULTI* allows multithreading in the outer loop; *N-SPEC* and *O-SPEC* are their speculative versions, respectively. For the inner loop, loop chunking is performed to create a maximum of 4 threads, each of which executes 50 iterations. The loop is always given 4 TPUs, including the master and the slaves.

The speculation applied in the parallel loop iterations does not increase the program speedup over the non-speculative execution because, within each outer loop iteration, the parent region is very small compared to the (speculated) child region. As a result, there is little computation to perform in parallel with the speculative one and the program suffers from the multithreading overheads involving both loop parallelisation and control speculation.

Figure 5.32 illustrates the restructuring of the outer loop. It is unrolled 4 times, followed by upward code motion so that the original parent regions of all the branches in a new unrolled iteration are packed together. The branches are predicted at the start of every outer loop iteration. There are several permutations in which the speculation can be performed. In Figure 5.31, *SPEC.1*, *SPEC.2*, *SPEC.3*, and *SPEC.4* speculate the first 1, 2, 3, and 4 branches respectively and in the order that they are encountered by the sequential flow of control. Furthermore, in order to restrict the TPU utilisation, the inner loop is always executed sequentially.

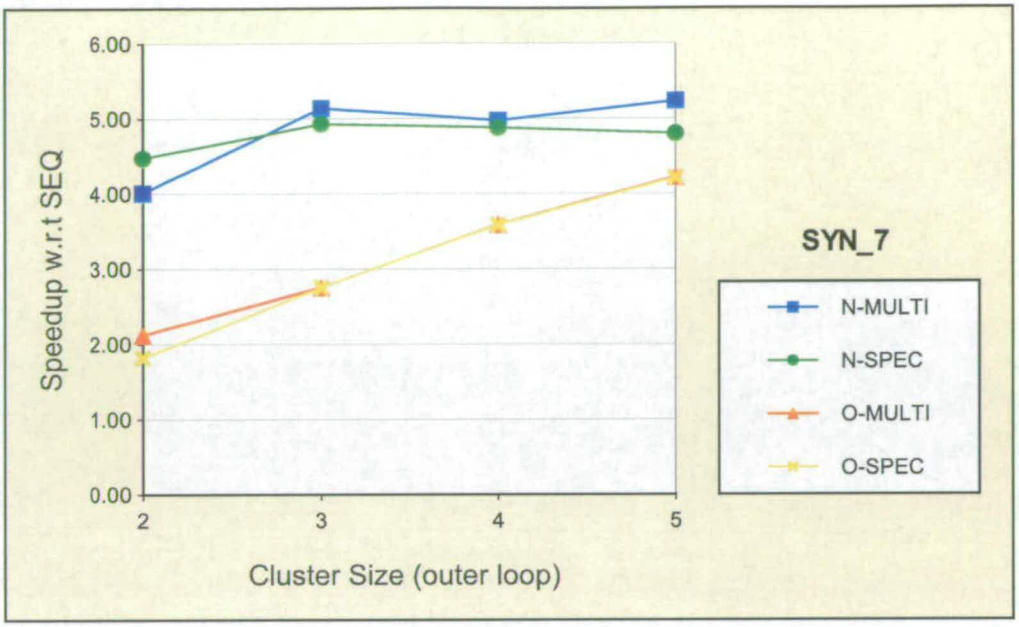


Figure 5.30: Speedup of multithreaded execution, with and without concurrent speculation

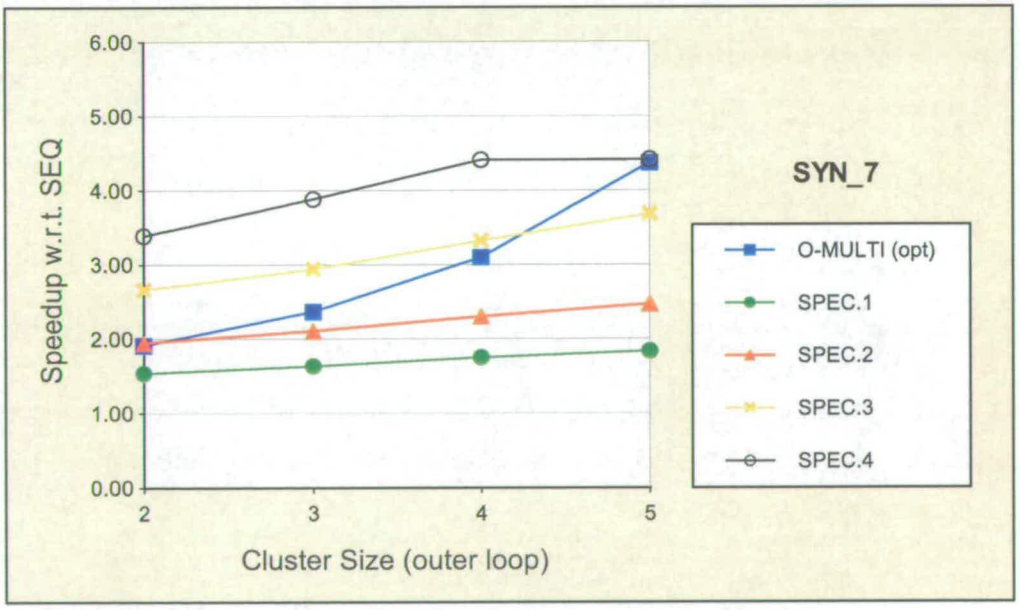


Figure 5.31: Speedup of speculative programs after the outer loop is optimised

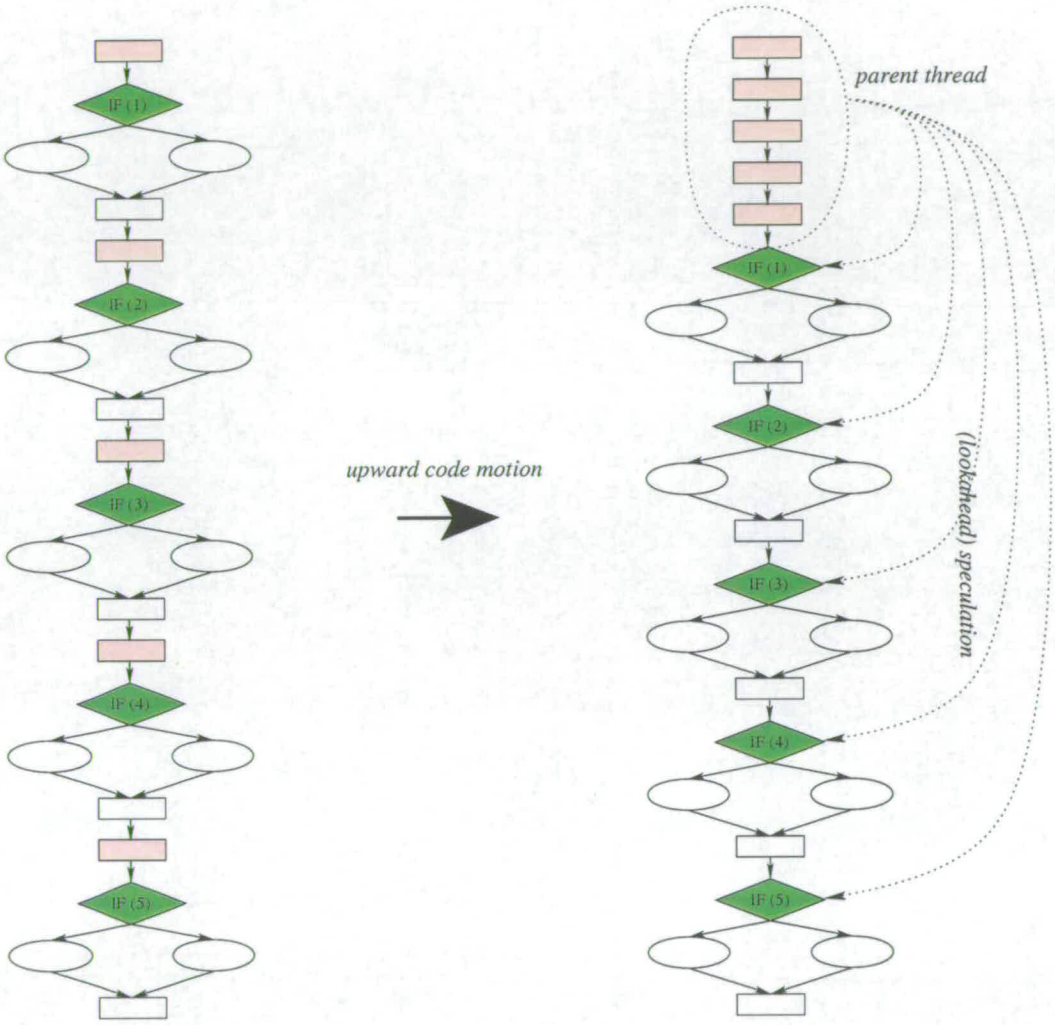


Figure 5.32: Loop unrolling and code motion being applied to the outer loop

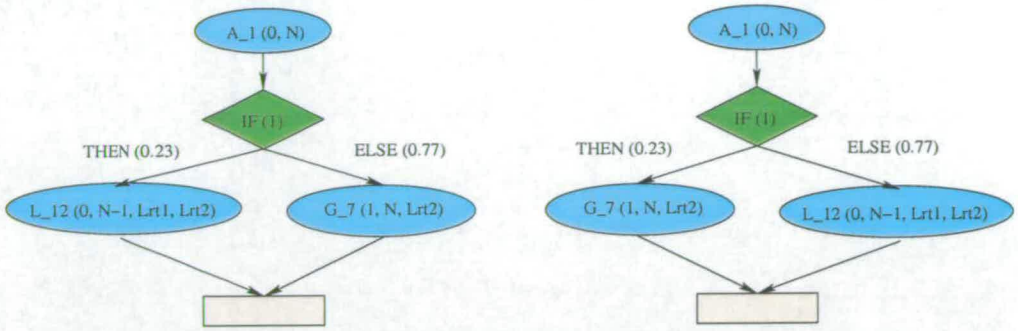
Comparing the speedup of *O-MULTI* in Figure 5.30 and *O-MULTI (opt)* in Figure 5.31, the optimised program performs slightly worse. On the other hand, the speedup of the speculative execution increases, particularly when at least 4 branches are predicted. *SPEC.4* and *N-MULTI* (in Figure 5.30) require similar amount of TPUs since each thread executing an outer loop iteration is assisted by 4 other threads. A comparison of speedup from both suggests that the available TPUs are still better used for the loop parallelisation of the inner loop than for the speculation.

5.2.2.4 Path Selection

In all the benchmarks considered so far, both paths of a branch contain identical substructures or identical loops (with the same sizes but different parameters). The branch probability was sufficient for choosing a path to be speculatively executed in the case of single-path speculation. However, for an unbalanced control structure, the path with a much higher workload albeit lower probability could be more critical.

Two synthetic benchmarks are displayed in Figures 5.33(a) and (b). Their control structures are similar to that of *SYN_1*, but loop *G* in one path is replaced by loop *L*. Loop parallelisation is the same as before, i.e. each loop is transformed for the multithreaded execution of 5 threads (including a master and slaves), each of which executes a maximum of 101 iterations. The contribution factor of each loop is also calculated and shown in Table 5.5.

In *SYN_UB_1*, it is obvious that loop *G* is on the more probable path and dominates the total execution time. Based on the previous observations (Section 5.2.2.1), it would be beneficial to speculate on this path and allocate the largest number of TPUs to this loop. In *SYN_UB_2*, although the ELSE path has the higher probability, the loop on this path contributes the least to the total execution time. Figure 5.34 shows the results



(a) *SYN_UB_1* (b) *SYN_UB_2*

Figure 5.33: Synthetic benchmarks with unbalanced control structures

Table 5.5 Contribution of each loop in *SYN_UB_1* and *SYN_UB_2*

Benchmark	Loop { *path* }	%	Loop { *path* }	%
SYN_UB_1	A {*control indep*}	28		
	L {*THEN*}	5	G {*ELSE*}	67
SYN_UB_2	A {*control indep*}	44		
	G {*THEN*}	32	L {*ELSE*}	24

from the speculative execution in *SYN_UB_2* as the following options are explored:

- Single-path speculation on the ELSE path (which contains loop L).
- Dual-path speculation (L+G).
- Single-path speculation on the THEN path (which contains loop G).

Since there can be at most 2 loops being executed at the same time, every loop is allocated 5 TPUs. It appears that when there are sufficient TPUs for all the loops, dual-path speculation yields the best speedup. In the case of single-path speculation, by executing loop G earlier instead of loop L, the speedup increases significantly and is

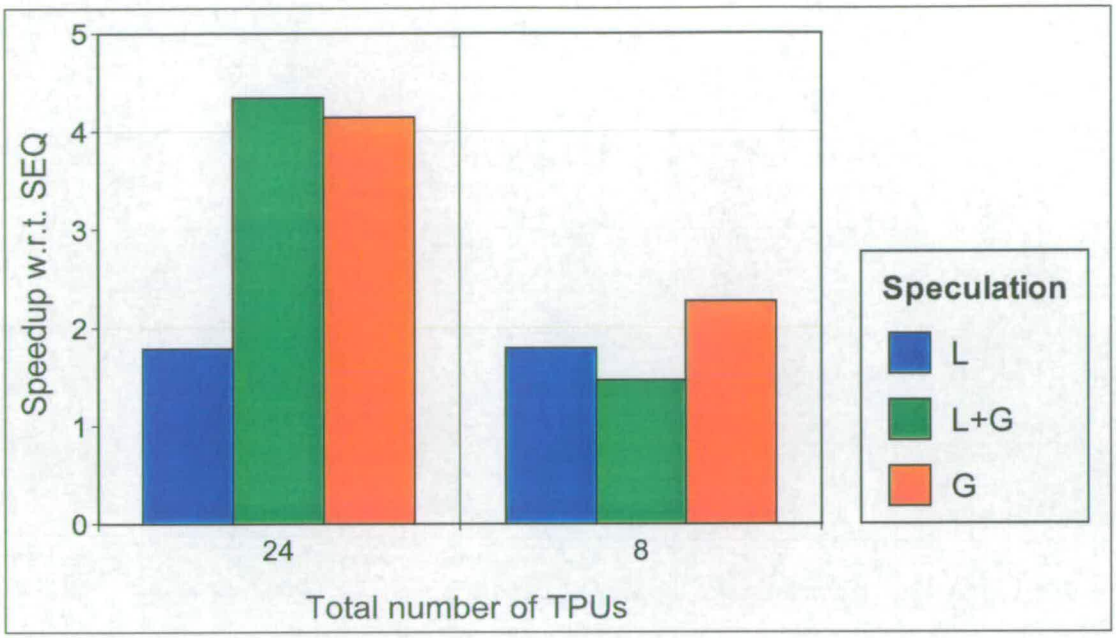


Figure 5.34: Speedup of the speculative execution in *SYN_UB_2*

almost as good as the result from dual-path speculation. However, when the number of TPUs is reduced to 8, dual-path speculation gives the worst speedup. The speculation on the THEN path (loop *G*) still gives better speedup than on the ELSE path (loop *L*), but the difference in their performance is small.

In this experiment, the order in which threads are forked in dual-path speculation is not significant. The reason is that loops *L* and *G* are both executed by multiple threads if there are 24 TPUs, or by single threads each if there are only 8 TPUs.

5.2.3 Summary

The effects of control speculation were studied using synthetic benchmarks comprising sets of parallelisable loops and conditional branches. First, by giving similar TPU allocation to the loops in both multithreaded non-speculative and speculative programs, the

latter performed slightly better than the former. Because the control speculation permits simultaneous execution of loops in several program fragments, the performance can be affected by poor resource allocation.

Empirical studies on TPU allocation schemes were conducted. Contribution of each individual loop to the overall program execution was computed using cumulative probability along the control-flow path until the loop is encountered and together with its sequential execution time. Among concurrently-executed loops, favouring the most dominant one (i.e. *Critical* strategy) delivered the best or close-to-best speedups. However, if none of those loops significantly contributed to the total execution time, better results were achieved by favouring the non-speculative one (i.e. *NonSPEC* strategy). Allocating the same number of TPUs to every loop (i.e. *All* strategy) yielded the best speedup only if there were not too many loops competing for the TPUs. If both paths of a branch have significantly different workload, then the contribution factor of each path which had been calculated for the resource allocation purpose can be used to determine which one could benefit more from the speculative execution.

Although multiple loops are executed simultaneously, they can be initialised by the cluster allocation commands at different cycles; the loop which is most favoured by the compile-time analysis may be the last one to acquire the TPUs. Furthermore, the loops whose execution have completed may free their TPUs a few cycles late. Since these are unforseen at the compile-time, the compiler should be aware that the total resource utilisation is slightly below the total resource availabilty, in order to avoid cluster and/or fork failures at the run-time.

Performing speculation in multiple levels of nested branches at the same time could be detrimental to the program performance as a result of resource contention. At best, the speedup achieved was only as high as the result from outermost-branch speculation.

The speculation in the deeper nest levels was less profitable due to the lower cumulative probabilities of the inner branches.

Besides the speculative execution of control-dependent threads (CSP), code fragments below the branches' re-convergences can be executed by control-independent threads (CI). The combination of both generally performed better than the use of only CSP. However, there are a few instances where it performed worse: when loops on the CI paths were too small and/or the CI threads predicted further trivial branches, while the CSP threads failed to allocate clusters to execute more critical program fragments.

Finally, while multiple iterations of a loop were executed in parallel, predicting the branches within those iterations provided improvement over the non-speculative execution, if there is sufficient parallel computation to offset the overheads of both loop parallelisation and control speculation. If the loop in question is an outer loop in a nest, the results so far suggested that the availble TPUs were better used for the multithreaded execution of the inner loop than for the control speculation.

5.3 Chapter Summary

Transformation modules for control-speculative execution were implemented using SUIF framework. They support single-path, dual-path, and nested speculation. This chapter also described the use of profile information and the pre-transformation analysis. Experiments were conducted to study the effectiveness of the control speculation, its interaction with multithreaded loop execution, and cluster allocation strategies. The execution of a control-independent path which post-dominates the branch was also examined.

Chapter 6

Conclusions

6.1 Thesis Summary

A framework has been proposed for multithreaded execution, which combines distributed program analyses, hierarchical thread management, and dynamic clustering of TPUs. The underlining idea is explained as follows. At compile-time, a program is repeatedly divided into sub-problems, each of which is specifically optimised and transformed by a class of compilation techniques. The subsystems and their finer partitions are organised in a hierarchy with master/slave relationships between them. During run-time, the master threads attempt to allocate clusters of slave TPUs on which the slave threads execute. The dynamic cluster allocation enables the utilisation of TPUs to be adjusted to the sub-problems' requirements throughout the program execution.

During the course of the research, a generic multithreaded architecture was modelled and simulated, which was inspired by CMP-based architectures such as Superthreaded. Enhancements were made to support hierarchy and dynamic cluster allocation, with the TPUs being equipped with special units that manage the threads'

parent/child and master/slave relationships. Furthermore, control speculation and register forwarding mechanisms were introduced. The main focus of the thesis was on compiler-based thread manipulation and the interface between the compiler and the architecture. The compiler plays an important role in exposing parallelism and orchestrating how programs will be executed on a relatively simple architecture. It requires commands, inquiries, and feedback to be passed between these two layers via specially-proposed instructions augmented to the MIPS instruction set. In addition to the architectural design and simulation, a multithreaded compilation package was implemented as a part of the SUIF compiler system. The package is composed of front-end transformers for the multithreaded loop and control-speculative execution, and a target-machine code generator.

With up to 16 TPUs, the multithreaded loop execution delivered speedup between 5 and 10 when combined with loop unrolling and loop peeling. This was achieved by dispatching the iterations to threads one-by-one in single-level multithreading. For nested loops, chunks of iterations were dispatched in order to restrict per-thread initialisation, synchronisation and retirement overheads, particularly for the inner loops. Speedups of around 4 or 5 were achieved. However, when this was applied to single-level multithreading, loop-level parallelism was compromised.

In the presence of conditional branches, speculative execution of the control- dependent paths boosted program speedup. The branches' post-dominating regions can be included into the speculative paths, aided by code motion and multithreaded transformation, in order to increase the thread granularity. Alternatively, when there is more parallelism to be exploited, control-independent threads can be launched to execute those regions. Speedup was generally further improved after both control-dependent speculation and control-independent execution were applied.

As several master threads simultaneously execute program fragments, or parallelisable loops in our benchmarks, they compete for available TPUs in order to allocate slave clusters. Cluster allocation strategies have impacts on the program performance. In the case of multiple loops being executed concurrently, best results were achieved by alloting the dominant ones greater number of TPUs. The loops' contribution to the overall program execution time was calculated using their sequential execution profiles and the branch probability profile.

Finally, the speedups achieved suggest that the benefit of the control speculation augments the gains made by loop parallelisation. In the experiments, both multithreaded non-speculative and speculative programs were generated using the same compilation options, i.e. parameters regarding loop unrolling, loop peeling, or loop chunking were the same for both versions so that their performance could be fairly compared. There are still outstanding issues such as: given a number of TPUs unused by loop parallelisation, should the compilation choices (including resource allocation) be further explored to improve the loop parallelisation instead of allocating the TPUs for speculative execution ? This will be discussed in the next section, along with some suggestions for future work.

6.2 Discussion and Future Works

6.2.1 Multithreaded Architecture

Like other CMP-based architecture [26, 32, 58, 66, 70], our multithreaded architecture is kept simple and relies heavily on the compiler to detect and exploit thread-level parallelism. A novel feature of the architecture is that clusters of TPUs are statically allocated to program partitions at compile-time, and this information is communicated to

the run-time system. The resource partitioning idea was inspired by SMT-based architectures [3, 44, 45, 71]. But these SMTs rely on a complete run-time system whereas, in our architecture, commands are passed from the compiler to perform cluster allocation. The framework brings an advantage of the SMT's philosophy to the CMPs, i.e. during the execution of a program, TPUs can be used in proportion to the amount of thread-level parallelism in each program partition and/or the priority given to these partitions (e.g. a non-speculative partition may be given more TPUs than a speculative one, if they are executed in parallel). The other main features and restrictions in the architecture are discussed next.

In practice, the location of slave TPUs on the chip and the size of the clusters would affect program performances differently. Large clusters are less likely to be successfully allocated than smaller ones and their TPUs are more likely to be scattered. The distance between slave TPUs, in reality, would impact the signal delays and communication between threads. The main data transmission is usually in the thread initialisation phase, where current register values are copied from the parent's register file to the child's. This operation would be more expensive than in other systems [3, 45, 58, 60], where the architectures consist of processing units arranged in ring topology (the child thread typically starts on the next processing unit in the ring) and registers can be rapidly transferred between *physically* neighbouring units. Besides the register transfer during thread initialisation, our register forwarding is similar to Multiscalar [12]. However, unlike Multiscalar, which propagates the forwarded registers to all the processing units, our architecture only forwards registers from the parent to the child threads. Because physically neighbouring TPUs could be assigned to different logical clusters, and these clusters may execute program partitions that are independent of each other, propagating the registers to all the TPUs would result in unnecessarily

high communication overhead. Issues such as VLSI implementation, hardware-level cluster allocation, and register communication mechanisms should be further studied.

Hardware support for speculative execution is kept to a minimum. A speculative buffer was added in each TPU to keep results when a thread is in speculative mode, with a mechanism to retrieve the correct version of the data from its predecessor. A new mechanism was developed to manage the speculative buffers according to the hierarchy of threads. Unlike other systems [32, 54, 65, 67], there is no hardware support for misspeculation detection and recovery as these are managed in the software. Also, the memory hierarchy (including caches) and data speculation were not included in our framework. To integrate these features into the architecture would require further investigations on how they would be organised around the hierarchy of threads. Other well-studied features, such as local branch predictors, should also be included to complete the functionality of the multithreaded architecture.

During the research, a restriction on the current architecture was noted: clusters do not operate entirely independently of each other. If there are several clusters simultaneously active, then only the one whose master TPU hosts the current head thread is able to reuse its slave TPUs. In the other clusters, the TPUs cannot be freed until the synchronisation signal is received by the master threads and passed on to the slaves. Our solution was to assign large chunks of program partitions to the slave threads, at the expense of compromising some parallelism. An alternative approach could employ multiple levels of synchronisation and allow each cluster to be operated using a unique signal. There are a few concerns with this idea. Firstly, a thread must distinguish the signal it uses as a master thread (e.g. when executing `crels`, it passes the signal to the slaves and waits until the signal returns) from the one it uses as a slave (e.g. when executing `xstp`, it waits for the signal from the master or the predecessor

slave). As multiple threads from several clusters may commit to the shared memory and retire simultaneously, care must also be taken to ensure that the program semantics is preserved.

Further improvements can be made to cluster formation and forking mechanisms. In this work, a thread checks for available TPUs and a value, *success* or *fail*, is returned before it proceeds to execute the next instruction. Superthreaded [68, 69, 70] employs a different approach called *delayed forking* which always successfully spawns a new thread in spite of the delay. However, if there is no TPU available over a long period, it may be better to let the current thread execute the code instead of waiting to spawn a new one. Alternatively, a time-out can be set for cluster formation and forking operations, with mechanisms that permit polling of available TPUs.

6.2.2 Multithreaded Compiler

Like other CMP-based systems such as Hydra [31, 32, 53, 54], STAMPede [65, 66, 67], or Superthreaded [68, 69, 70, 79], our multithreaded compiler was developed specifically for the proposed architecture - it is aware of the execution models supported and the restrictions in the architecture when generating multithreaded programs. In these compilers, program transformations are performed at the front-end, where high-level program structures such as loops can be easily recognised. Loop parallelisation is a main feature presented in all the compilers. Multiple loop iterations are typically executed by multiple threads in parallel in a predecessor/successor style. However, threads in Hydra, STAMPede, and Superthreaded commit to the shared memory and retire in a sequential order. As each thread commits, it also updates the current state of the processor. In our system, slave threads (which also execute loop iterations in a predecessor/successor style) commit and update the cluster's state, maintained by the

master thread, instead of the processor's state. This hierarchical thread management would fit in well with the multithreaded execution in nested loops, provided that multiple clusters can be operated independently, as discussed in the previous section. With the current solution to dispatch big chunks of inner-loop iterations to slave threads, nested multithreading only performed as well as one-level multithreading in the outermost loops.

In addition to the loop parallelisation, our compiler also generates code for coarse-grained control speculation. Like STAMPede's approach, the compiler inserts instructions to mark speculative regions in the program and threads can switch between non-speculative and speculative execution. However, unlike STAMPede, the misspeculation detection, and recovery actions are all managed by software routines. Furthermore, because data speculation is not supported, threads will be forced to wait until the data they depend upon is made available. Our multithreaded compiler would therefore have to detect and work around data dependencies between threads (the benchmarks used in the research have only few data dependencies).

The multithreaded code generator is a modification of a MIPS code generator. Several back-end analysis were therefore not specifically targeted at our multithreaded architecture. Because register usage is known after register allocation is performed at the back-end, register forwarding has to be handled separately from the multithreaded transformation at the front-end. User's specifications are needed to specify which program partitions will use the register communication instead of the default memory communication. This differs from Multiscalar's approach [72], in which both task selection and register communication are performed at the assembly-code level. Consequently, the Multiscalar compiler only needs to perform control-flow and data-flow analysis once. Our compiler, on the other hand, has to perform the analysis in both the

front-end and the back-end compilation.

Besides the features mentioned above, there are still restrictions in the current compiler, as discussed next. The SUIF compiler system [84] was well suited for the implementation of the compiler prototype because various specialised functions can be implemented separately and communicate via internal program representation and annotations. Basic structures, such as procedures, loops, and conditional branches, are easily recognised from the internal format. Hence, distributed program analysis and compilation would be well supported. More functions are still required to make the system fully automatic. At present, the compiler does not perform inter-procedural analysis and it relies on built-in SUIF functions to detect data dependencies (ones that are not detected by the compiler can be specified via a graphical user interface tool). However, in some areas such as embedded applications, where the compilation is performed only once, the current semi-automatic system can still be useful provided that the code is specifically compiled and fully optimised to achieve high performance.

For the multithreaded loop execution, heuristics or analytic models should be developed to estimate performance trends and determine a point where the benefit from loop-level parallelism peaks or reaches a plateau. After this point, further use of the control speculation could be worthwhile. Another use of heuristics or analytic models is in cluster allocation scheduling. As optimal requirement for individual program partitions can be estimated, schedules for the availability and the utilisation of TPU resources can then be determined for the entire application. One concern is the level in which analytic models are used in the compiler. Costs estimated from the high-level and the low-level internal representation might be very different, such that, after the transformation based on the front-end analysis, the final output programs may behave differently than expected. Feedback loops would be needed for the compilation pro-

cess, as described in Chapter 3. Methods for mapping costs or results from the analysis in multiple levels should also be implemented.

Another limitation in this work is the use of profile information: the programs being profiled, analysed, and transformed, always used the same input data and setting. More insight could be gained by using a variety of benchmarks that execute different input data sets. Further work is required for the simulator to accommodate larger and more realistic benchmarks, and very importantly, the integration of operating system and library calls into the simulator package.

6.2.3 Applications

One type of program that can be tackled by the compiler and executed on the multithreaded architecture are loop-based ones. There can be data dependence between loop iterations, and the dependency is detected by the SUIF compiler. In the compilation flow described in Chapter 4, basic loop optimisations such as loop normalisation, loop skewing, and loop reversal were also performed by the SUIF compiler prior to the multithreaded loop parallelisation to rearrange bounds and data dependency pattern in the loops. In the case of control speculation, the control structures need to be quite large. There should be substantial amount of computation in both the parent threads (which execute the code before conditional branches) and the child threads (which execute the speculated paths of the branches). Furthermore, the amount of data dependencies between program partitions should be at minimum. If possible, the program partitions should be independent from each other.

Numeric programs such as Livermore [81, 82] used in the research would fit well in the framework, and examples that could be related to the synthetic benchmarks used in Chapter 5 are scientific calculators. In particular, signal and image processing for

multimedia applications [25, 74], which are loop intensive (and the loop iterations are largely independent), would benefit from this approach. In these applications, a number of threads could also perform several computations in parallel, and communicate to each other but not regularly.

6.3 Conclusion

The main contribution of this thesis is the experimental evaluation of hierarchical multithreading in a framework consisting of a simulated multithreaded architecture and a compiler. Within the framework, fragments of a program can be specifically optimised and executed by clusters of thread processing units (TPUs) as orchestrated by compile-time analysis. A multithreaded processor architecture has been proposed, which supports dynamic clustering of the TPUs and speculative execution. The transformation from sequential programs into multithreaded ones are performed in the compiler. The focus was on multithreaded loop and control-speculative execution. Based on the experimental results, significant program speedups were achieved by loop parallelisation, and could be further improved by control speculation.

Appendix A

Examples of Control-Flow Graphs

The experiments in Chapter 5 used synthetic benchmarks which are well-structured. However, in real applications, some control-flow graphs would need pre-processing before they can be transformed for multithreaded control-speculative execution. The benchmarks used for demonstration in this chapter are *heapsort* [1] and *164.gzip* [83].

A.1 heapsort

This program performs heap-sorting on 2000 elements of an array in the ascending order. A control-flow graph (CFG) of the sorting function is shown in Figure A.1. This benchmark was not use in the experiments in Chapter 5 because its control structures are too fine, i.e. a child region of each branch except *IF(3)* contains an average of 3 instructions (counted in the C-code level). The child region of *IF(3)* contains a small sequential loop which checks and swaps between elements of the array.

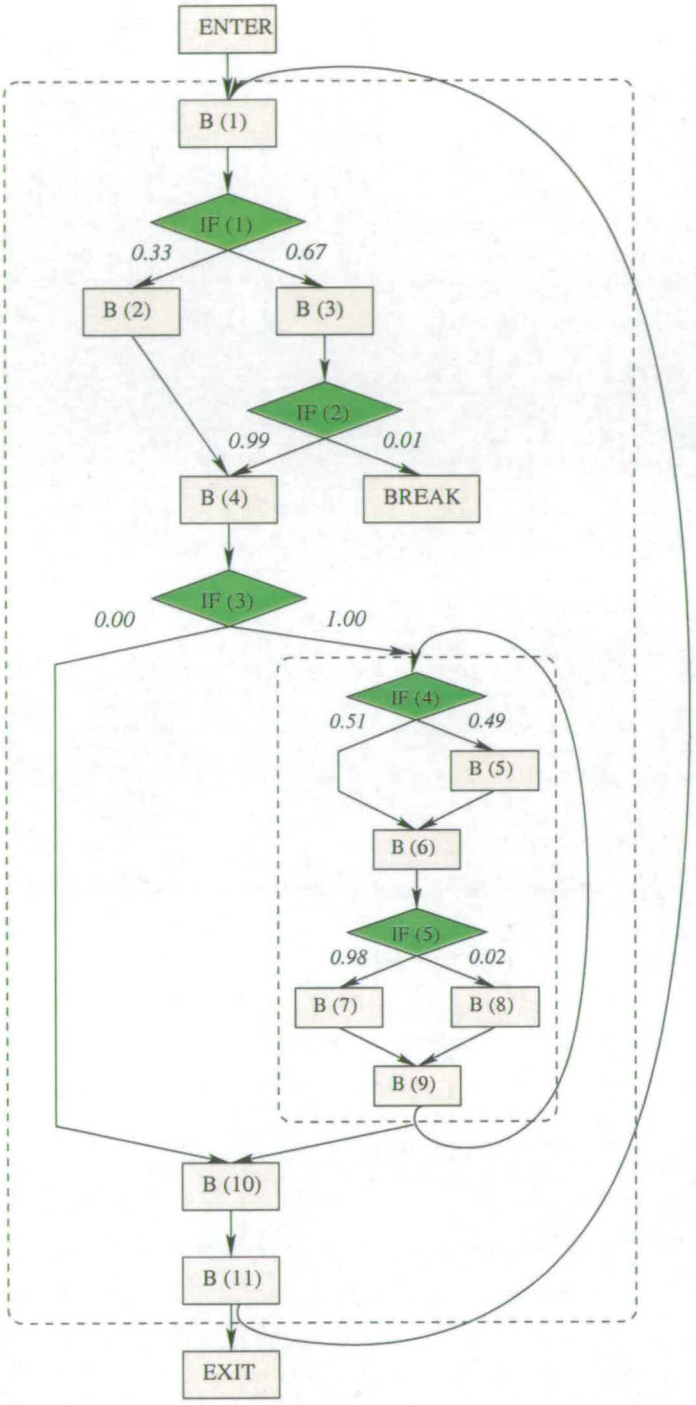


Figure A.1: CFG of the heap-sorting function

Suppose that the code granularity is not an issue, possible options for applying the control speculation on this CFG are:

1. **Speculation on IF(3)**

   In this case, the parent region of *IF(3)* may include all the predecessor nodes according to the forward control flow, which are node *B(1)* and structures *IF(1)* and *IF(2)*. The child region is the loop containing *IF(4)* and *IF(5)*. Since there is a *BREAK* in the parent region, the child thread must be aborted if the parent executes this instruction (see Section 5.1.1, the handling of control-flow breaks).

2. **Speculation on IF(1) and IF(2)**

   IF(2) is an incomplete sub-structure of *IF(1)* as their child regions are overlapped. Code replication technique can be applied so that each region has a separate copy of node *B(4)* and structure *IF(3)*, as shown in Figure A.2. There is a *BREAK* in the child region of *IF(2)* which is not allowed because the child thread cannot exit the loop before the speculation is resolved. Thus, it is replaced by setting a flag *cont* to *FALSE*. This flag is set to *TRUE* when a new iteration starts, and is evaluated either after the speculation is resolved (at node *B(13)*) or along with the loop continuation test at the end of that iteration (at node *B(11)*. The structures *IF(1)* and *IF(3)* form a series of branches while *IF(2)* is nested inside *IF(1)*.

   The structure *IF(3)* may also be replicated, as shown in Figure A.3, in order to increase the size of the speculative threads. In this new CFG, *IF(3)* and *IF(3')* can be speculated in conjunction with *IF(1)* and *IF(2)*.

   The nested-speculation template was described in Section 5.1.3, and tested in *SYN_5* and *SYN_6* benchmarks. Furthermore, the handling of a series of specu-

lative structures were tested in *SYN_3* and *SYN_4* benchmarks.

3. **Speculation on IF(4) and IF(5)**

   IF(4) and *IF(5)* form a series of branches which are embedded in a sequential loop. *IF(4)* has only one child region which is low confident. This path may be speculated if it contains a large program partition, as discussed in Section 5.2.2.4. Otherwise, *B(6)* which is a post-dominating, control-independent node may be executed instead. The control-independent execution was described and evaluated in Section 5.2.2.2. On the other hand, single-path speculation can be applied to *IF(5)*.

   However, suppose that the loop is parallelisable, the TPU resources may be better dedicated to the loop parallelisation than to the control speculation. This was discussed in Section 5.2.2.3.

4. **Speculation on IF(3) and IF(4)-IF(5)**

   There is a loop boundary between the structure *IF(3)* and the series of *IF(4)* and *IF(5)*. Thus, *IF(3)* and *IF(4)-IF(5)* do not fit into our nested-speculation template. An even more complicated case is if the loop is parallelised and concurrent speculation (Section 5.2.2.3) is performed. However, these have not been studied in the thesis. In such cases, the compiler only speculates on *IF(3)*.

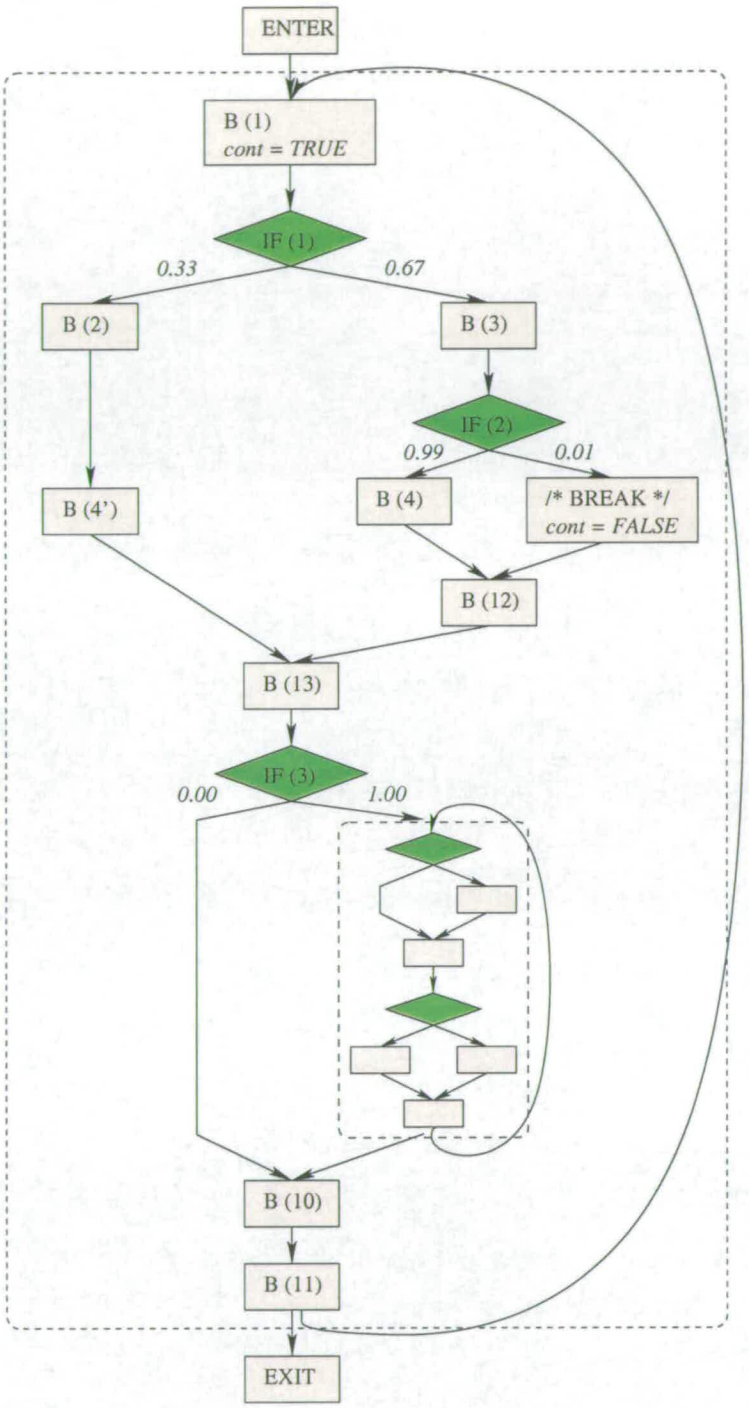


Figure A.2: CFG of the heap-sorting function after code replication (1)

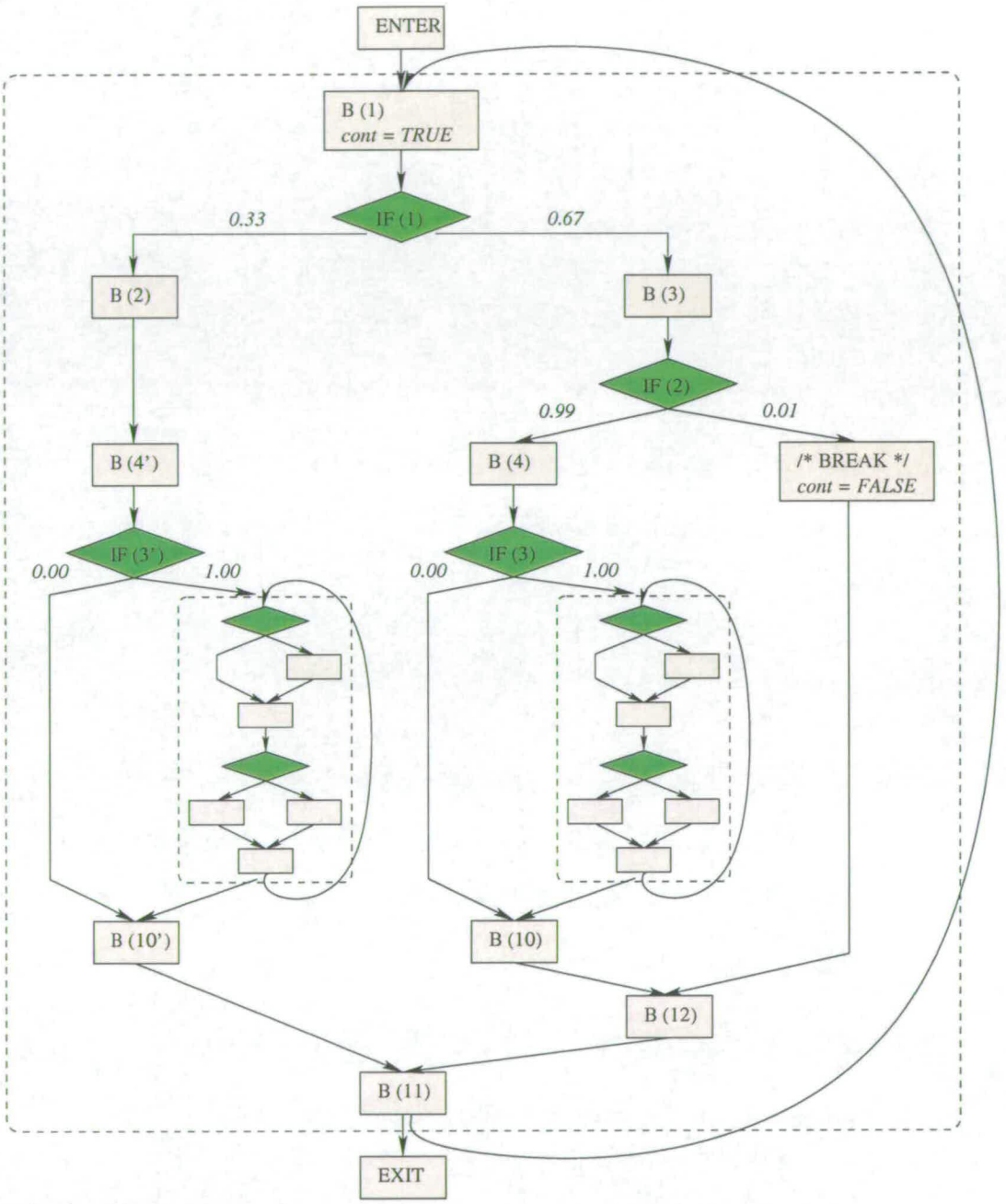


Figure A.3: CFG of the heap-sorting function after code replication (2)

A.2 164.gzip

This benchmark was taken from the SPEC CPU2000 suit [83]. It performs data compression and decompression. The size of the benchmark is too big for both the multi-threaded compiler and the simultor. Therefore, it was only profiled using *test* data set and the control-flow graph was constructed manually. Figures A.4 and A.6 show the control-flow graphs of procedures *deflate_fast* and *inflate_block*, respectively.

1. **Function deflate_fast**

   The control-flow graph of *deflate_fast* consists of a series of branches, some of which are nested ones. To handle function calls, the compiler would inline them, if possible, since it does not perform inter-procedural analysis. The functions cannot contain instructions that raise exceptions or cause program exit; otherwise, an exception-free version of them are generated. From the example, *longest_match* contains assertion tests that cause the program to exit if the assertions fail. Suppose that *IF(1)* is not speculated but is included in the parent region of *IF(3)*. Instead of exiting the program immediately, an invalid value is returned from *longest_match* (see Figure A.5). This value is checked at the caller's site after the call instruction; then the child thread is aborted and the parent thread exits the program.

   Similarly, *ct_tally* in the child region of *IF(3)* contains assertion tests. The compiler would normally avoid speculation on *IF(3)*. Nevertheless, suppose that the speculation is performed, a possible handling of *ct_tally* is shown in Figure A.5. An extra condition may be added after the function call to check the validity of the returned value. However, the program exit will be delayed until the speculation is resolved, i.e. at node *B(8)*.

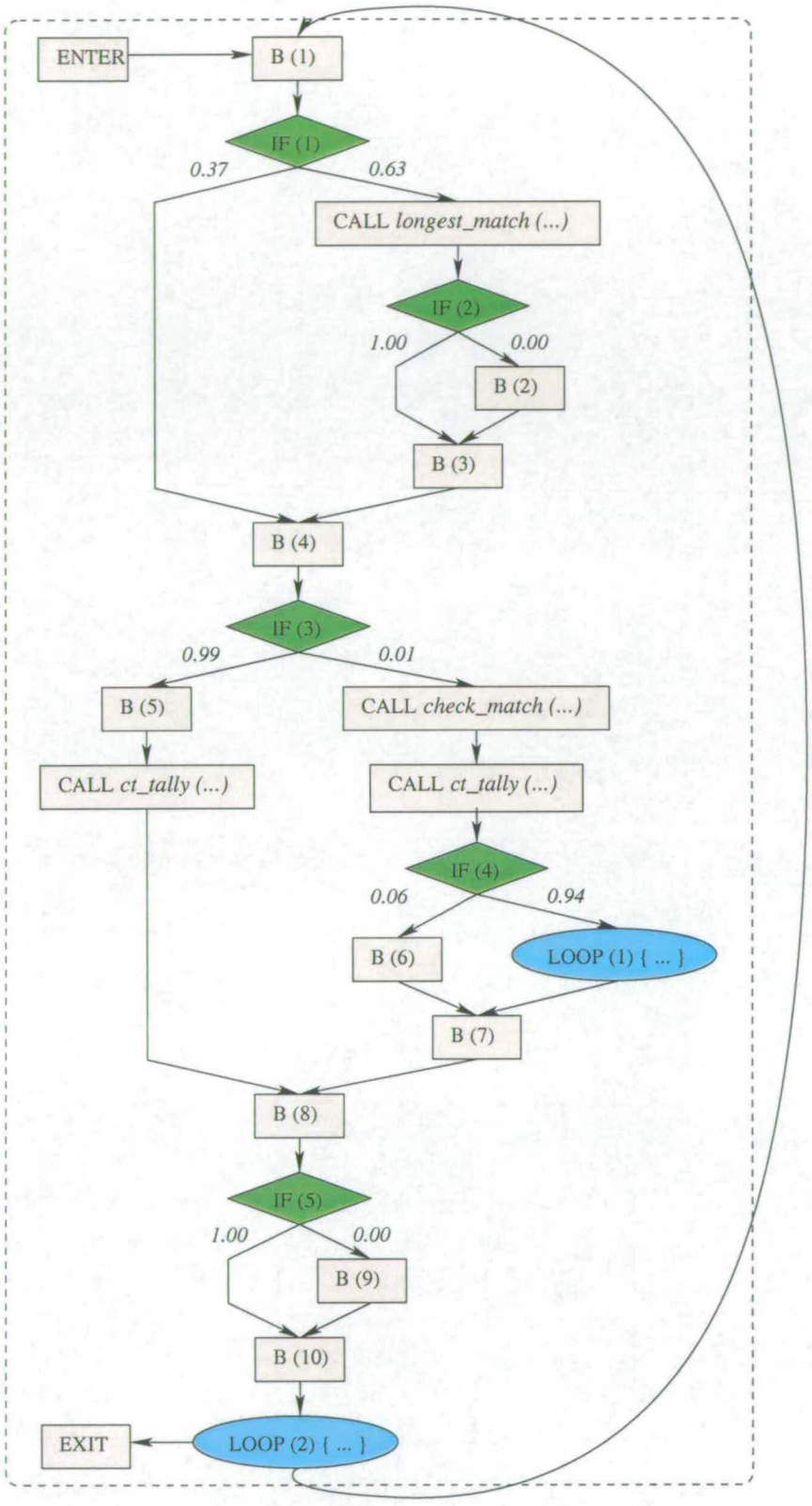


Figure A.4: CFG of procedure *deflate_fast*

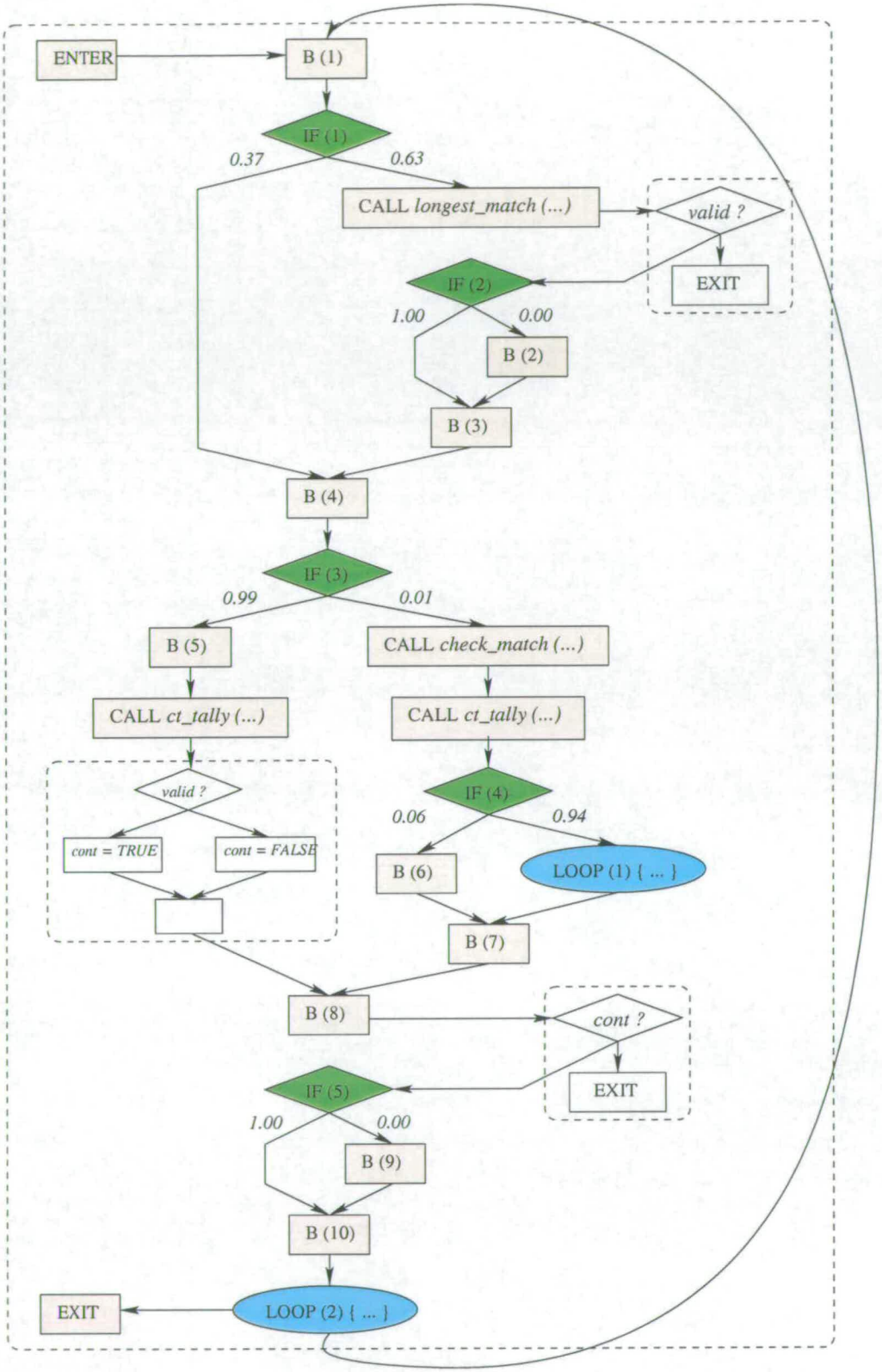


Figure A.5: Handling of function calls inside procedure *deflate_fast*

2. **Function inflate_block**

   In Figure A.6, suppose that functions *inflate_dynamic*, *inflate_stored*, and *inflate_fixed* do not contain exception instructions, program exits, or calls to any other functions. The branches *IF(1)* and *IF(2)* can therefore be speculated. However, while being speculative, a child thread that executes *inflate_dynamic* or *inflate_stored* cannot exit the speculation scope, i.e. function *inflate_block*. The SUIF compiler generates a temporary variable *tmp* to store a value returned from *inflate_dynamic*, *inflate_stored*, or *inflate_fixed*. After the outermost branch *IF(1)* is resolved, *tmp* can then be returned from *inflate_block*. Figure A.7 shows the branch structures which were arranged in a completely-nested form.

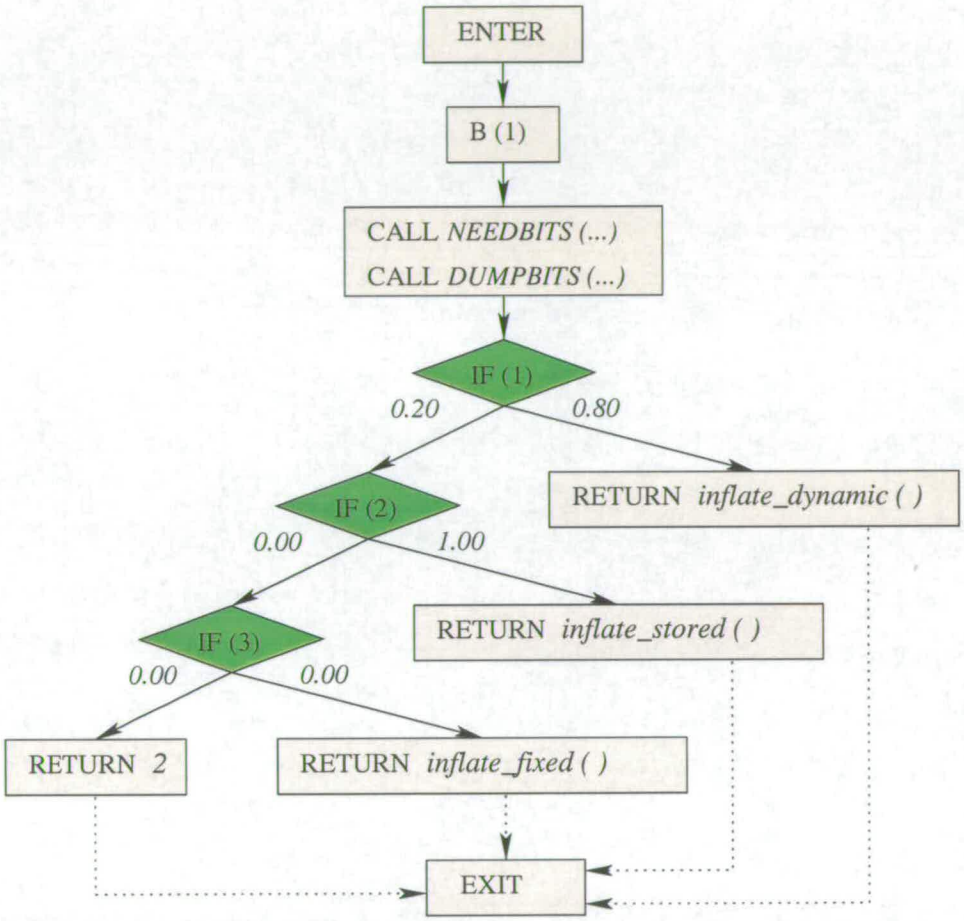


Figure A.6: CFG of procedure *inflate_block*

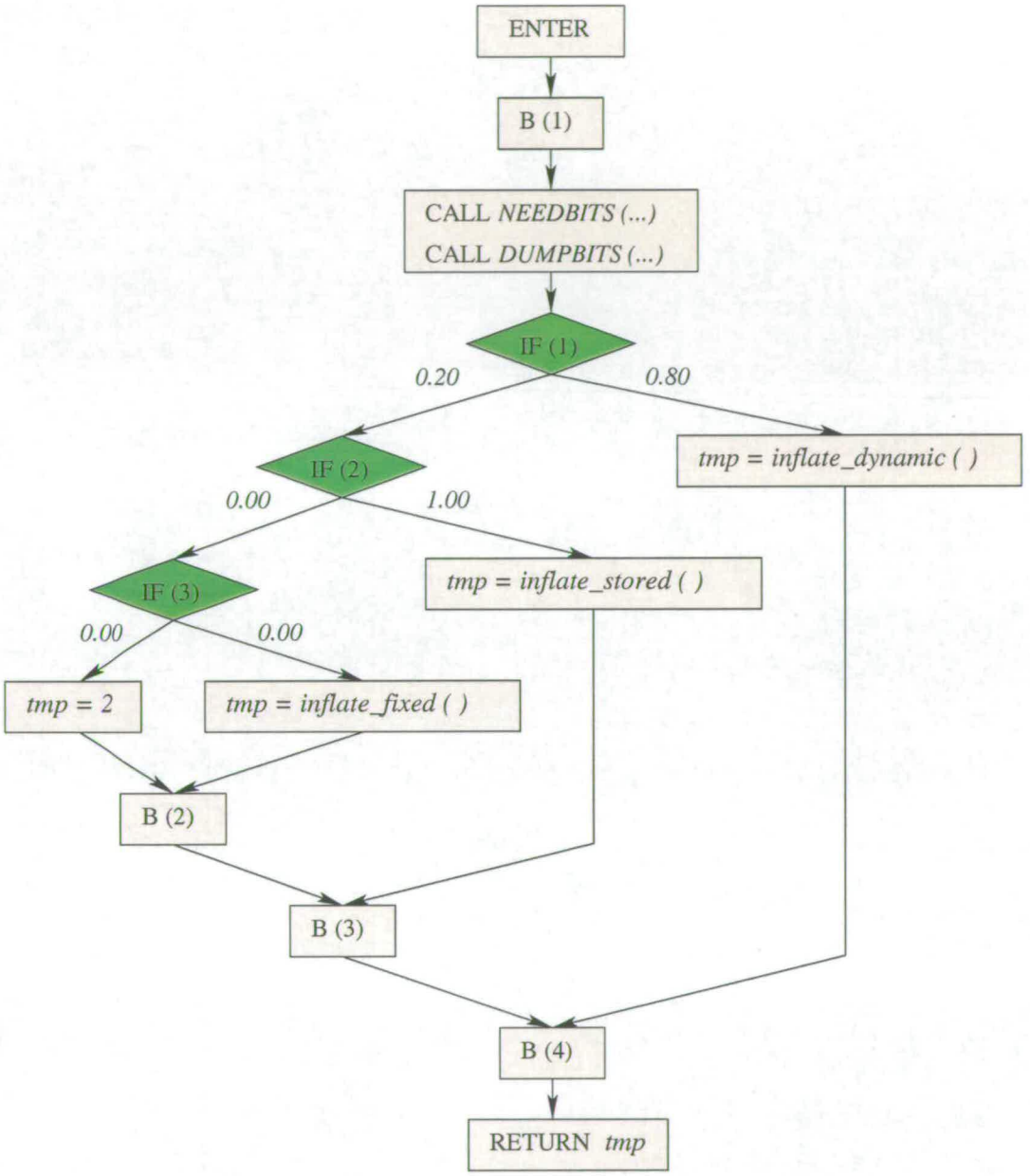


Figure A.7: Completely-nested branches in procedure *inflate_block*

Appendix B

Global Thread Control Unit

The Global Thread Control Unit (GTCU) is a central unit in the multithreaded processor architecture. It maintains threads' information including relative order of all the active threads and a pointer to the head thread. The GTCU is accessed quite often during multithreaded execution, and even more frequently during (multithreaded) speculative execution (see Sections 3.2.1 and 3.4.1.3 for details). The following experiment compares programs' performance when the access delay is set to 0, 1, and 2 time units (all the results in Chapters 4 and 5 are based on zero delay). The benchmarks used are Livermore kernels from Chapter 4, which were transformed as follows:

1. Non-speculative programs.

   The benchmarks were transformed using **Loop_Transformer_1**. Results from these programs, with the GTCU's access delay being set to zero, were shown in Figure 4.11.

2. Speculative programs.

   The benchmarks were transformed using **Loop_Transformer_2**. These are the ones mentioned in Page 84.

Both versions of the multithreaded programs generate a lot of threads during run-time as the TPUs are reusable, and the speculative programs access the GTCU more often than the non-speculative ones due to the speculative load/store operations. Results are shown in Figures B.1 and B.2. Speedup of the non-speculative programs when the GTCU delay is 0, 1, and 2 time units are very close. For the speculative programs, the difference in speedup is slightly more pronounced (when the delay is set to zero, the speculative programs give very similar speedup to the non-speculative ones).

It appears that the delay in the GTCU has only slight impact on program's performance because most accesses to the GTCU are for reading thread sequence and this unit is managed in a multiple-readers/single-writer style. To avoid contention and long access delay in a centralised unit (for the multiple reads), a table could be implemented with each entry being a copy of the thread sequence exclusively used by each TPU, thus allowing multiple read operations to proceed in parallel. A write operation, on the other hand, will lock the whole table as it needs to broadcast an update in one entry to all the others. This should not cause bottleneck in the system since each thread updates the GTCU table only twice, i.e. when it is forked and when it retires, although it may read from the GTCU table several times during its execution while perfoming speculative load/store operations.

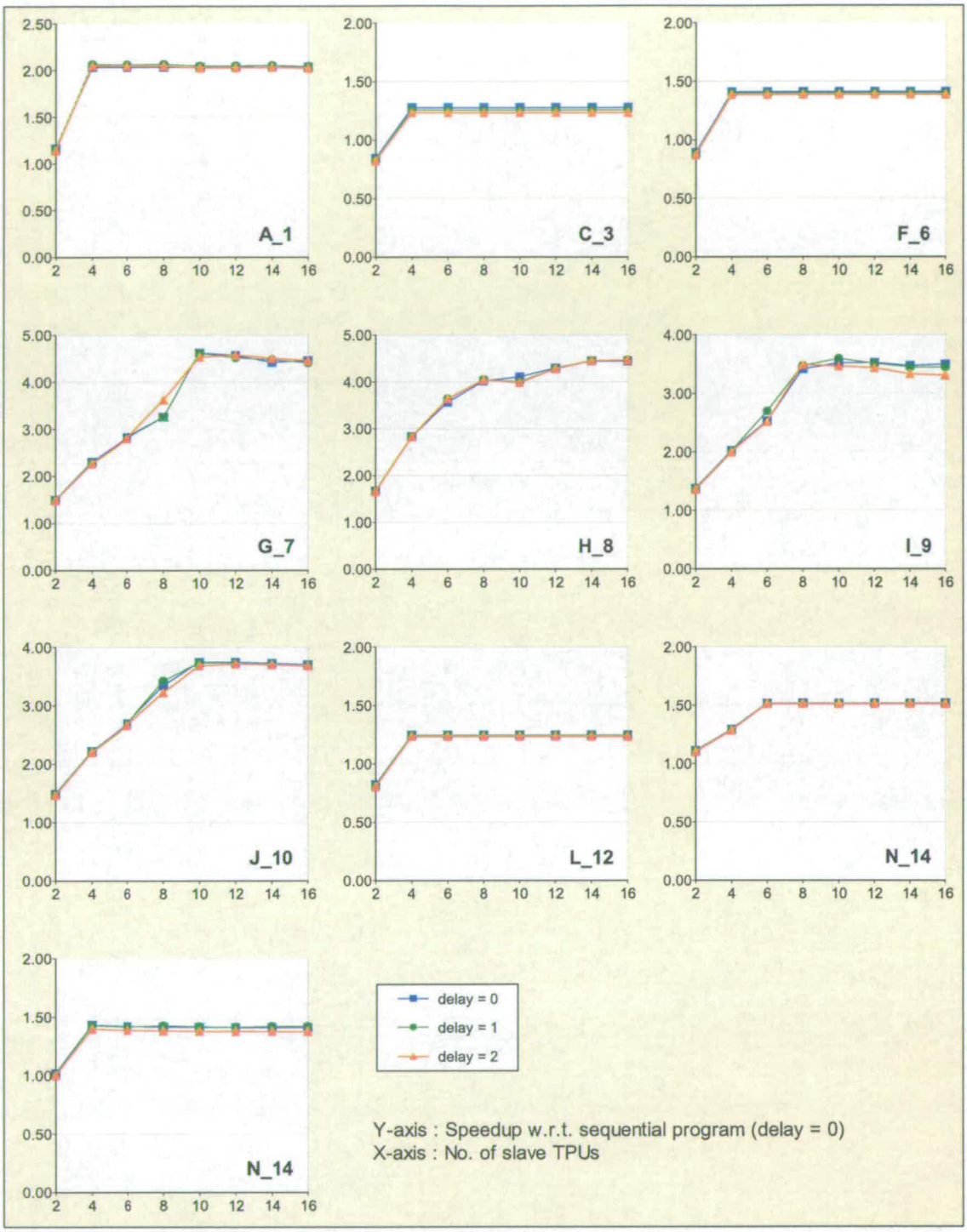


Figure B.1: Speedup of non-speculative programs (with GTCU delay = 0, 1, and 2 time units)

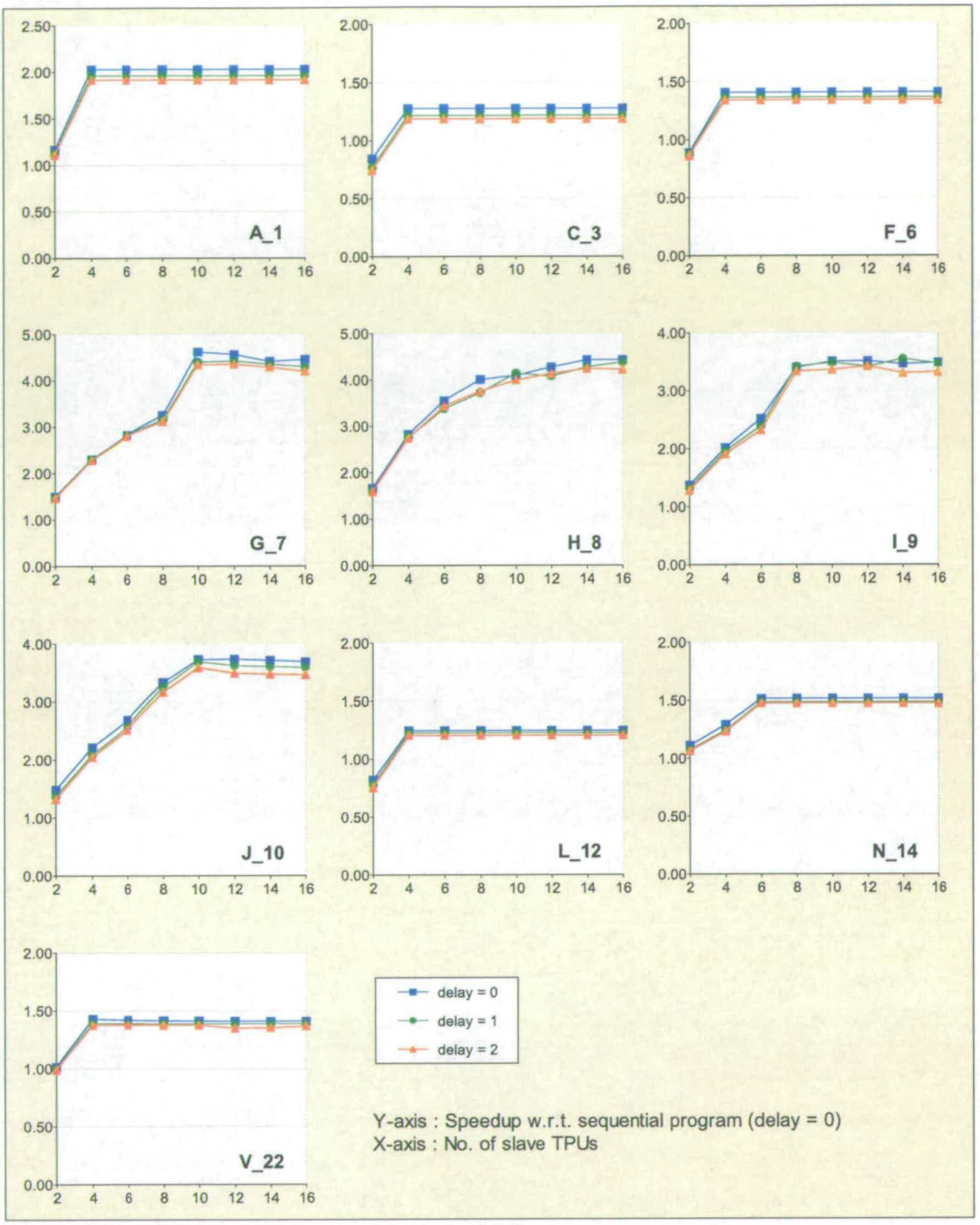


Figure B.2: Speedup of speculative programs (with GTCU delay = 0, 1, and 2 time units)

Bibliography

[1] Alfred Aburto. FTP site. Naval Oceans Systems Center. ftp://ftp.nosc.mil/pub/aburto.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing, 1986.

[3] Haitham Akkary. *A Dynamic Multithreading Processor*. PhD thesis, Electrical and Computer Engineering, Portland State University, 1998.

[4] Andrew W. Appel and Maia Ginsburg. *Modern compiler implementation in C*. Cambridge University Press, 1998.

[5] Christoffer Arvidsson. A multi-threaded architecture platform. Master's thesis, Division of Informatics, University of Edinburgh, September 1999.

[6] D. K. Arvind and R. Rangaswami. Asynchronous multithreaded processor cores for system level integration. In *Proceedings of the Conference on Intellectual Property (IP' 99)*, pages 105–110, Edinburgh, November 1999. Miller Freeman.

[7] Jean Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database, and Distributed Systems*, chapter 9: Low-level mechanisms for process synchronization, pages 210–248. Addison Wesley, 1993.

[8] Thomas Ball and James R. Larus. Branch prediction for free. In *Conference on Programming Language Design and Implementation*, pages 300–313, June 1993.

[9] Utpal Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[10] David Bernstein, Doron Cohen, and Hugo Krawczyk. Code duplication: an assist for global instruction scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, pages 103–113, November 1991.

[11] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, Bill Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Effective automatic parallelization with polaris. *International Journal of Parallel Programming*, May 1995.

[12] Scott E. Breach, T. N. Vijaykumar, and Gurindar S. Sohi. The anatomy of the register file in a multiscalar processor. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, November 1994.

[13] Brad Calder, Dirk Grunwald, and Amitabh Srivastava. The predictability of libraries. Technical Report WRL Technical Note TN-50, Western Research Laboratory, July 1995.

[14] Ben Catanzaro. *Multiprocessor System Architectures*, chapter 8: Multithread Programming Facilities for Implementing Multithreaded Applications, pages 229–268. SunSoft Press, 1994.

[15] Robert S. Chappell, Jared Stark, Sangwook P. Kim, Steven K. Reinhardt, and Yale N. Patt. Simultaneous Subordinate Microthreading (SSMT). In *Proceedings of the 26th International Conference on Computer Architecture*, pages 186–195, May 1999.

[16] William Y. Chen, Scott A. Mahlke, Nancy J. Warter, Sadun Anik, and Wen-mei W. Hwu. Profile-assisted instruction scheduling. *International Journal for Parallel Programming*, 22(2):151–181, April 1994.

[17] Gautham K. Dorai and Donald Yeung. Transparent threads: Resource sharing in SMT processors for high single-thread performance. In *Proceedings of the 11st International Conference on Parallel Architectures and Compilation Techniques (PACT 2002)*, Virginia, September 2002.

[18] Predeep K. Dubey, Kevin O'Brien, Kathryn M. O'Brien, and Charles Barton. Single-program speculative multithreading (SPSM) architecture: Compiler-assisted fine-grained multithreading. In *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques (PACT 95)*, pages 109–121, June 1995.

[19] John R. Ellis. *Bulldog: a compiler for VLIW architectures.* MIT Press, 1986.

[20] Hesham El-Rewini and Hesham H. Ali. How many times should A loop be unrolled ? In *Proceedings of the 7th Intl. Conf. Parallel and Distributed Computing Systems*, Las Vegas, October 1994.

[21] Keith I. Farkas, Norman P. Jouppi, and Paul Chow. Register file design considerations in dynamically scheduled processors. Technical Report WRL Research Report 95/10, Western Research Laboratory, November 1995.

[22] Erin Farquhar and Philip Bruce. *The MIPS Programmer's Handbook.* Morgan Kaufmann, 1994.

[23] Marco Fillo, Stephen W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-machine multicomputer. Technical Report AIM-1532, Laboratory of Computer Science, MIT, 1995.

[24] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th Annual International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 85–95, MA, USA, October 1992.

[25] James D. Foley, Andires van Dam, Steven K. Feiner, John F. Hughes, and Richard L. Phillips. *Introduction to Computer Graphics.* Addison-Wesley Publishing Company, 1994.

[26] Manoj Franklin. *The Multiscalar Architecture.* PhD thesis, University of Wisconsin-Madison, 1993.

[27] Manoj Franklin and Gurindar S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers*, May 1996.

[28] Eric Freudenthal and Alan Gottlieb. Process coordination with fetch-and-increment. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 260–268, Santa Clara, CA, April 1991.

[29] Milind Girkar, Mohammad R. Haghighat, Paul Grey, Hideki Saito, Nicholas J. Stavrakos, and Constantine D. Polychronopoulos. Illinois-Intel multithreading library: Multithreading support for intel architecture based multiprocessor systems. *Intel Technology Journal, Q1 Issue*, February 1998.

[30] Sridhar Gopal, T. N. Vijaykumar, James E. Smith, and Gurindar S. Sohi. Speculative versioning cache. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA 4)*, February 1998.

[31] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, San Jose, CA, October 1998.

[32] Lance Hammond, Benedict A. Hubbert, Michael Siu, Manohar K. Prabhu, Michael Chen, and Kunle Olukotun. The Stanford Hydra CMP. *IEEE MICRO*, pages 71–84, March-April 2000.

[33] Timothy Heil and J. E. Smith. Selective dual path execution. Internal technical report, Department of Electrical and Computer Engineering, University of Wisconsin-Medison, November 1996.

[34] Japheth E. Hossell. Compiling java byte code for multithreaded architecture. Master's thesis, Division of Informatics, University of Edinburgh, September 1999.

[35] Quinn Jacobson, Steve Bennett, Nikhil Sharma, and James E. Smith. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd Inter-*

national Symposium on High Performance Computer Architecture (HPCA 3), Texas, February 1997.

[36] Quinn Jacobson, Eric Rotenberg, and Jim Smith. Path-based next trace prediction. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, December 1997.

[37] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[38] Venkata Krishnan and Josep Torrellas. A clustered approach to multithreaded processors. In *Proceedings of the International Parallel Processing Symposium*, pages 627–634, March 1998.

[39] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computer, Special Issue on Multithreaded Architecture*, September 1999.

[40] Jee Myeong Ku. The design of an efficient and portable interface between a parallelizing compiler and its target machine. Master's thesis, Electrical Engineering, University of Illinois at Urbana-Champaign, 1995.

[41] Bil Lewis and Daniel J. Berg. *Threads Primer: A Guide to Multithreaded Programming*, chapter 5: Synchronisation, pages 61–86. SunSoft Press, 1996.

[42] Mikko H. Lipasti and John Paul Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual International Symposium on Microarchitecture (MICRO-29)*, December 1996.

[43] Mikko H. Lipasti. *Value locality and speculative execution*. PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University, 1997.

[44] Jack Lee-jay Lo. *Exploiting thread-level parallelism on Simultaneous Multithreaded processor*. PhD thesis, University of Washington, 1998.

[45] Pedro Marcuello, Antonio Gonzalez, and Jordi Tubella. Speculative multithreaded processors. In *Proceedings of the ACM International Conference on Supercomputing (ICS 98)*, Australia, 1998.

[46] Pedro Marcuello and Antonio Gonzalez. Control and data dependence speculation in multithreaded processors. In *Proceedings of the Workshop on Multithreaded Execution, Architecture, and Compilation (MTEAC 98)*, 1998.

[47] Pedro Marcuello and Antonio Gonzalez. Clustered speculative multithreaded processors. In *Proceedings of the ACM International Conference on Supercomputing (ICS 99)*, Greece, 1999.

[48] Pedro Marcuello, Jordi Tubella, and Antonio Gonzalez. Value prediction for speculative multithreaded architectures. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32)*, November 1999.

[49] Pedro Marcuello and Antonio Gonzalez. A quantitative assessment of thread-level speculation techniques. In *Proceedings of the 1st International Parallel and Distributed Processing Symposium*, Mexico, May 2000.

[50] John M. Mellor-Crummey and Michael L. Scott. Scalable reader-writer synchronization for shared-memory multiprocessors. In *Proceedings of the 3rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 106–113, Williamsburg, Virginia, 1991.

[51] Frank Mueller and David B. Whalley. Avoiding unconditional jumps by code replication. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 322–330, June 1992.

[52] Tarun Nakra, Rajiv Gupta, and Mary Lou Soffa. Global context-based value prediction. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA 5)*, Florida, January 1999.

[53] Kunle Olukotun, Lance Hammond, and Mark Willey. Improving the performance of speculatively parallel applications on the Hydra CMP. In *Proceedings of the*

ACM International Conference on Supercomputing (ICS 99), Rhodes, Greece, June 1999.

[54] Jeffrey Oplinger, David Heine, Shih-Wei Liao, Basem A. Nayfeh, Monica S. Lam, and Kunle Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University Computer Systems Laboratory, February 1997.

[55] Alastair Patrick. A co-design environment for java programs targetting asynchronous processors. Bachelor's thesis, Division of Informatics, University of Edinburgh, June 1999.

[56] William Pugh. A practical algorithm for exact array dependence analysis. *Communication of the ACM*, 35(8):102–114, August 1992.

[57] Martin Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 112–123, Las Vegas, NV, 1997.

[58] Eric Rotenberg, Quinn Jacobson, Yiannakis Sazeides, and Jim Smith. Trace processors. In *Proceedings of 30th Annual International Symposium on Microarchitecture (MICRO-30)*, December 1997.

[59] Eric Rotenberg and Jim Smith. Control independence in trace processors. In *Proceedings of the 32th Annual International Symposium on Microarchitecture (MICRO-32)*, November 1999.

[60] Eric Rotenberg. *Trace Processors: Exploiting Hierarchy and Speculation*. PhD thesis, University of Wisconsin-Madison, 1999.

[61] Radu Rugina and Martin Rinard. Design-driven compilation. In *Proceedings of the International Conference on Compiler Construction*, Genova, Italy, 2001.

[62] Hideki Saito, Nicholas Stavrakos, Steven Carroll, Constantine Polychronopoulos, and Alex Nicolau. The design of PROMIS compiler. In *Lecture Notes in Computer Science 1575*. Springer Verlag, March 1999.

[63] Yiannakis Sazeides and James E. Smith. The predictability of data values. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, December 1997.

[64] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 4th Annual International Symposium on Computer Architecture*, volume SIGARCH Newsletter 9(3), pages 135–148, May 1981.

[65] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Architecture support for thread-level data speculation. Technical Report CMU-CS-97-188, School of Computer Science, Carnegie Mellon University, November 1997.

[66] J. Gregory Steffan and Todd C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture (HPCA 4)*, Las Vegas, February 1998.

[67] J. Gregory Steffan, Christopher B. Colohan, and Todd C. Mowry. Extending cache coherence to support thread-level data speculation on a single chip and beyond. Technical Report CMU-CS-98-171, School of Computer Science, Carnegie Mellon University, December 1998.

[68] Jenn-Yuan Tsai. *Superthreading: Integrating Compilation Technology and Processor Architecture for Cost-Effective Concurrent Multithreading*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1998.

[69] Jenn-Yuan Tsai, Zhenzhen Jiang, Zhiyuan Li, David J. Lilja, Xin Wang, Pen-Chung Yew, Bixia Zheng, and Stephen J. Schwinn. Superthreading: Integrating compilation technology and processor architecture for cost-effective concurrent multithreading. *Journal of Information Science and Engineering, Special Issue on Compiler Techniques for High-Performance Computing*, 14(1):205–222, March 1998.

[70] Jenn-Yuan Tsai, Jian Huang, Christoffer Amlo, David J. Lilja, and Pen-Chung Yew. The superthreaded processor architecture. *IEEE Transactions on Computers, Special Issue on Multithreaded Architectures and Systems*, 48(9), September 1999.

[71] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, Italy, June 1995.

[72] T. N. Vijaykumar. *Compiling for the Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, 1998.

[73] Staven Wallace, Brad Calder, and Dean Tullsen. Thread multiple path execution. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[74] Alan Watt and Mark Watt. *Advanced Animation and Rendering Techniques: Theory and Practice*. ACM Press, 1992.

[75] Michael Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman Publishing, 1989.

[76] Michael Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

[77] Mark N. Yankelevsky and Constantine D. Polychronopoulos. α-Coral: A multigrain, multithreading processor architecture. In *Proceedings of the International Conference on Supercomputing (ICS 01)*, pages 358–367, 2001.

[78] Mohamed M. Zahran and Manoj Franklin. A feasibility study of hierarchical multithreading. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[79] B. Zheng, J. Y. Tsai, B. Y. Zang, T. Chen, B. Huang, J. H. Li, Y. H. Ding, J. Liang, Y. Zhen, P. C. Yew, and C. Q. Zhu. Designing the Agassiz compiler for concurrent multithreaded architectures. In *Workshop on Languages and Compilers for Parallel Computing*, August 1999.

[80] Craig B. Zilles, Joel S. Emer, and Gurindar S. Sohi. The use of multithreading for exception handling. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32)*, 1999.

[81] The ASCI LFK benchmark code.
http: //www. llnl. gov/asci_benchmarks/asci/limited/lfk/asci_lfk.html.

[82] Livermore loops coded in C. http://www.netlib.org/benchmark/livermorec.

[83] Standard Performance Evaluation Corporation (SPEC).
http://www.specbench.org/.

[84] The SUIF 1.x compiler system. http://suif.stanford.edu/suif/suif1/.

[85] TCOVSUIF 2.0. http://brass.cs.berkeley.edu/tcovsuif2.html.